Scaling Up Transactions with Slower Clocks

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Andreia Correia
University of Neuchâtel
andreiacraveiroramalhete@gmail.com

Abstract

Concurrency controls with optimistic read accesses and pessimistic write accesses are among the fastest in the literature. However, during write transactions these algorithms need to increment an atomic variable, the central clock, limiting parallelism and preventing scalability at high core counts.

In this paper, we propose a new concurrency control, Deferred Clock Transactional Locking (DCTL), which significantly reduces the heartbeat of the central clock, thus increasing scalability. DCTL will not increment the clock for consecutive disjoint transactions. An optimized variant, named DCOTL, allows for consecutive transactions with non-disjoint write-accesses to commit without incrementing the clock. Moreover, we show variants of these two algorithms with starvation-free transactions.

Transactions in DCTL are opaque, which means it can be applied to concurrent data structures, Database Management Systems, Software Transactional Memory, and Persistent Transactional Memory. Our experiments show that these DCTL algorithms match or surpass the current state of the art for most workloads. We adapted both algorithms using an existing durability technique and implemented a fully transactional DBMS with disk persistence, whose scalability in write transactions exceeds the current state of the art.

CCS Concepts • **Theory of computation → Concurrent algorithms**.

Keywords concurrency control, starvation-freedom, software transactional memory

ACM Reference Format:
Pedro Ramalhete and Andreia Correia. 2024. Scaling Up Transactions with Slower Clocks. In *The 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '24), March 2–6, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3627535.3638472

PPoPP '24, March 2–6, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0435-2/24/03…$15.00
https://doi.org/10.1145/3627535.3638472

1 Introduction

Multiple concurrency controls with opaque transactions exist in the literature. Some of the most efficient are based on a combination of pessimistic synchronization for write accesses with optimistic for read accesses. This is the case of the TL2 [11] and LSA [18, 19] algorithms, both acquiring locks for write accesses, while read accesses require pre and post-validation during transactional execution, and commit-time validation of the read-set. At the end of a write transaction, in TL2 and LSA the central clock is advanced with an atomic increment-and-fetch and the number returned is the *commit timestamp*, which is then placed on the sequence-locks as they are being released, to indicate the timestamp of the latest modification to their respective data.

Previous work [43] has shown that despite their success, TL2 and LSA hit a scalability bottleneck in write intensive transactional workloads, with the reason being the contention generated when incrementing the central clock. Concurrency controls such as TicToc [54], 2PLSF [43] and PLOR [8] have shown that reducing, or altogether foregoing the increment of the central clock, can significantly increase scalability for short write transactions. However, previous techniques have had to pay a cost to achieve this. In the case of 2PLSF and PLOR, read accesses are pessimistic, which are slower than the optimistic accesses of TL2 and LSA, making 2PLSF and PLOR less efficient on read-mostly workloads. Moreover, pessimistic locking incurs more traffic in the cache-coherence system which is detrimental to scalability in servers with a high core count [52]. As for TicToc, read accesses are optimistic but transactions are not opaque, which means it cannot be deployed in generic code, nor easily used to implement concurrent data structures. Other solutions like Cicada [31] use the hardware clock, and deal with clock skew using novel techniques.

These observations led us to the following question: *Is it possible to design an opaque concurrency control with a centralized atomic variable as the transaction clock, while achieving high scalability for write-heavy workloads?*

The answer to this question is *yes*. We present a new concurrency control, named **D**eferred **C**lock **T**ransactional **L**ocking (DCTL), which makes it possible to commit write transactions without incrementing the clock. At the core of DCTL lie three insights: that multiple disjoint in-flight transactions can share the same commit timestamp, that the timestamp can be shared even on consecutive transactions which are disjoint in time, and consecutive transactions by

the same thread on the same memory locations do not require the timestamp to advance.

Sharing the commit timestamp reduces the need to increment the central clock, which increases scalability and throughput. On the other hand, non-disjoint consecutive transactions require the advance of the clock. When the clock is not incremented, consecutive transactions with accesses in the same memory locations will restart and increment the clock. Applying this technique to non-disjoint workloads, like queues, will not take advantage of deferring the clock increments. These algorithms assume that consecutive transactions will not execute on the same memory locations, deferring the clock advance to a later execution that accesses the same memory location. The core idea of DCTL is that we only need to *order* transactions which access the same memory locations, whether on read accesses or write accesses.

The second algorithm, DCOTL, goes beyond this idea, further minimizing the increment of the central clock by foregoing the need to order consecutive transactions with non-disjoint write-accesses. In DCOTL the clock must be incremented only if a read access follows a write access.

We also describe a generic technique to provide starvation-freedom to time-based concurrency controls, with a fallback to 2PL [4, 17] after a certain number of failed attempts.

In addition, our generic concurrency control algorithms can be combined with durability techniques to create a PTM (Persistent Transactional Memory) in particular, we took the state of the art PTM TL4x [1] and modified it to use our concurrency control. Due to the absence of a clock increment on every write transaction, several modifications were required to maintain correctness of the algorithm.

Our algorithms execute optimistic read accesses, with high scalability for write transactions, without sacrificing performance in read-mostly workloads, thus providing:

- High scalability for disjoint write transactions;
- High scalability for disjoint and non-disjoint read txns;
- Starvation-free transactions;
- Durable transactions on disk, which we utilized to create two scalable database engines, DCTL4x and DCOTL4x;

The rest of the paper is organized as follows. We first discuss related work in §2, then present our concurrency control DCTL in §3. We describe a starvation-free technique in §5. We summarize the modifications required to add buffered durability to DCTL in §6. We perform an evaluation of DCTL and its variants in §7, and finally conclude in §8.

2 Related Work

Time-based concurrency controls are among the highest scaling in the literature, possessing a central clock (an atomic shared variable) which is incremented at the end of every write transaction. They acquire sequence-locks for writing and use optimistic read accesses with commit time read-set validation. At commit time, after all write-locks have been acquired, a read-set validation procedure is performed to ensure that all data read during the transaction is consistent. To ensure opacity [23], at the beginning of the transaction the central clock is read to a thread-local variable `tl_readTS` which is used throughout the transaction to validate the timestamps of the sequence-locks, guaranteeing they are lower than `tl_readTS` for every read and write access, i.e modifications happened before the transaction started.

TL2 is a time-based concurrency control by Dice et al [11]. LSA is another time-based concurrency control [18], capable of adjusting the time window to have less conflicts. The LSA algorithm has been implemented in TinySTM [19] using an undo-log for better performance. TicToc [54] removes the bottleneck of incrementing a central clock in OCC (Optimistic Concurrency Control) by computing transaction timestamps lazily at commit time.

A variant of TL2, named GV5, reduces the number of clock increments [13]. For GV5, it is enough to have a conflict on any previously modified sequence-lock, for the clock to increment. Every time there is an allocation or de-allocation, this variant will trigger an unnecessary restart.

When it comes to read transactions, both TL2 and LSA have a lightweight read interposing function, which consists of reading the timestamp on the sequence lock, then reading the data, then reading the timestamp again, followed by reading a thread-local variable (`tl_readTS`), containing the timestamp taken at the start of the transaction. At the end of the read transaction there is nothing to be done, no increment of the central clock, nor validation of a read-set. These factors are the reason for the high scalability these algorithms can achieve on read transactions, with DCTL going one step further by foregoing the first read on the sequence-lock. TicToc on the other hand does not make a distinction between write transactions and read transactions, thus requiring the usage of a read-set and its validation at commit time, even for read transactions. This means that the throughput of TicToc is not as good as the other algorithms for read-mostly workloads (see Figure 9).

The idea of using loosely synchronized clocks has been explored by Cicada [31] in the context of in-memory DBMS and highlighted in previous work [47]. Cicada uses an optimistic concurrency control with scalable timestamp allocation using distributed (per-thread) clocks, allocated from the hardware clock TSC and deals with the corresponding clock skew with two novel techniques [31]. Its authors note that using traditional timestamp allocation with atomic fetch-and-add on a shared counter, drops throughput from 56.5 M tps to 6.2 M tps (millions of transactions per second). Cicada requires timestamp allocation and commit time read-set validation for read transactions, unlike TL2, LSA and DCTL.

Situated in an opposite corner of the design space, are fully pessimistic techniques like TLRW [12], 2PLSF [43] and PLOR [8]. These use two-phase locking, acquiring read-locks before read accesses and write-locks before write accesses.

Classical 2PL does not require a central clock, which means that scalability is theoretically high. Unfortunately, 2PL has two disadvantages over optimistic techniques. The *first* is the extra synchronization cost of acquiring a read-lock (at least one load-store fence) versus doing optimistic validation (one load-load fence). This cost is fixed, i.e. it does not affect the slope of the scalability plots however, it dominates the throughput in read-mostly workloads. The *second* disadvantage is the extra traffic in the cache-coherence system and contention stemming from the acquisition of the read-locks [52]. Lock contention can be reduced by using scalable read-indicators [6, 16, 30, 43] but it comes at an higher synchronization cost for write-lock acquisition and increased memory usage. The larger obstacle is the flooding of cache-coherence backbones caused by the acquisition of the read-locks, which optimistic techniques do not have. These disadvantages are shared by TLRW, 2PLSF, PLOR and likely any other pure 2PL-based technique. As we will show later, our starvation-free variants DCTL-SF and DCOTL-SF, do not suffer from these issues because they fallback to a 2PL technique only after K_f failed attempts in optimistic mode.

Starvation-free STMs have been shown previously in the literature [7, 21, 43, 46, 51].

3 Deferred Clock Transactional Locking

In this section we introduce the DCTL algorithm (**D**eferred **C**lock **T**ransactional **L**ocking), and an optimized variant named DCOTL.

Algorithm 1 shows DCTL and DCOTL, with code shown in blue for the starvation-free variants which will be explained later in Section 5. To ease exposition we encapsulated the differences between DCTL and DCOTL in two functions, named `isWriteConsistent()` and `isReadConsistent()`, whose implementations are shown in Algorithms 3 and 4, with `tl_*` indicating thread-local and `g*` global variables.

Both DCTL and DCOTL are optimistic concurrency controls, implying that lock acquisition is required only for write accesses, `stmWrite()`, and that read accesses, `stmRead()`, perform post-validation for the data read. In addition, write transactions must execute commit time read-set validation (line 28), based on a timestamp read at the start of the transaction (line 13). Transactional blocks are delimited by `beginTxn()` and `commitTxn()`. Our DCTL implementations utilize *type annotation* with C++ operator overloading to automatically trigger calls to `stmRead()` on read accesses, and `stmWrite()` on write accesses. Our implementations are undo-log based, however they could equally be done with a redo-log. We chose an undo-log because on redo-log approaches each read access implies a lookup on the redo-log data structure, which incurs a significant cost on read-mostly workloads. During transactional execution, an `stmRead()` or `stmWrite()` may read inconsistent data, or at commit time read-set validation may fail, and in these cases the transaction will automatically restart, calling `restartTxn()` (line 34).

Algorithm 1 Common code DCTL and its variants

```
1  TicketLock gTicketLock ;          // starvation-free irrevocable lock
2  atomic<uint64_t> gClock {0};                          // central clock
3  WriteSet tl_writeSet {};                    // append-only undo-log
4  ReadSet tl_readSet {};                       // append-only read-set
5  uint64_t tl_readTS {};                 // timestamp of the current txn
6  uint64_t tl_attempts {0};   // number of restarts of the current txn
7  RWSeqLock seqLock[NUM_LOCKS];  // array of sequence-locks

9  void beginTxn() {          // always inlined or preprocessor macro
10   setjmp();
11   tl_readSet.reset();
12   tl_writeSet.reset();
13   tl_readTS = gClock.load();       // all txns read the central clock
14   if (tl_txnType == IRREVOCABLE) gTicketLock.lock();
15 }

17 void commitTxn() {
18   std::atomic_thread_fence(std::memory_order_seq_cst);
19   if (tl_txnType == IRREVOCABLE) {
20     uint64_t commitTS = gClock.load();
21     for (rwlock : tl_writeSet) rwlock.unlock(commitTS);
22     for (rwlock : tl_readSet) rwlock.unlock();
23     tl_attempts = 0;
24     gTicketLock.unlock();
25     return;
26   }
27   if (tl_writeSet.size == 0) { tl_attempts = 0; return; }  // read txn
28   if (!tl_readSet.validate()) restartTxn();      // read-set validation
29   uint64_t commitTS = gClock.load();      // timestamp allocation
30   tl_writeSet.unlock(commitTS);  // release locks with commitTS
31   tl_attempts = 0;
32 }

34 void restartTxn() {
35   tl_writeSet.revertModifications();          // undo modifications
36   uint64_t commitTS = gClock.fetch_add(1);      // advance clock
37   tl_writeSet.unlock(commitTS);  // release locks with commitTS
38   tl_attempts++;
39   if (tl_attempts == K_fails) tl_txnType = IRREVOCABLE;
40   longjmp();                  // continues at setjmp() in beginTxn()
41 }
```

3.1 DCTL

The insight for reducing clock increments is the realization that a transaction does not need a unique commit timestamp. Furthermore, the central clock (`gClock`) is only required so that future transactions have a consistent view of previously committed transactions. This is enabled by assigning a timestamp `tl_readTS` at the start of each transaction (line 13), and validating on each access that the timestamp on the sequence-lock is lower than `tl_readTS`. Foregoing the increment of the clock at commit time may cause later transactions to have an inconsistent read access, as the sequence-lock will be *equal* to the `tl_readTS` of a later transaction, which in

Algorithm 2 Read and Write interposing functions

```
41 T stmRead(T* addr) {
42   if (tl_txnType == IRREVOCABLE) {
43     tl_readSet.add(addr);            // needed to unlock read-locks
44     seqLock[addr2idx(addr)].readLock();     // acquire read-lock
45     return val;                             // read data and return it
46   }
47   T val = *addr;                                          // read data
48   loadLoadFence();
49   uint64_t sl = seqLock[addr2idx(addr)].load();     // read seqlock
50   if (!isReadConsistent(sl)) restartTxn();
51   if (tl_txnType == WRITE_TXN) readSet.add(addr);
52   return val;
53 }

55 void stmWrite(T* addr, T newValue) {
56   if (tl_txnType == IRREVOCABLE) {
57     seqLock[addr2idx(addr)].writeLock();    // acquire write-lock
58     tl_writeSet.add(addr, *addr);           // save the previous data
59     *addr = newValue;                       // write the new data
60     return;
61   }
62   uint64_t sl = seqLock[addr2idx(addr)].load();     // read seqlock
63   if (!isWriteConsistent(sl)) restartTxn();
64   if (!seqLock[addr2idx(addr)].tryWriteLock()) restartTxn();
65   tl_writeSet.add(addr, *addr);             // save the previous data
66   *addr = newValue;                         // write the new data
67 }
```

turn triggers a restart of that transaction. Once a restart occurs, the clock will be incremented (line 36), otherwise the transaction would continuously restart.

DCTL postpones the increment of the clock to the moment when the transaction detects an inconsistent access. This novel approach allows for multiple transactions to share the same commit timestamp, substantially reducing the heartbeat of the central clock, thus minimizing contention on the global clock variable. This is the main reason for the gains we will show in Section 7. Furthermore, *consecutive* transactions which execute one after the other *and* access completely disjoint data, can also share the same commit-timestamp. As a consequence, if the dataset has N memory locations, and we assume each transaction accesses M disjoint memory locations, we can have N/M consecutive disjoint transactions use the same commit timestamp.

Instead of incrementing the central clock at the end of every write transaction, DCTL defers this increment, at the risk of a later transaction seeing inconsistent data and restarting. At first sight, it may seem like a poor design choice to prefer restarting the transaction versus incrementing a shared variable, given the relative cost of restarting a transaction to be higher than the cost of an uncontended atomic increment, but there are two reasons why restarting is preferable.

The first reason is that when a restart occurs, it reduces the throughput of *one* individual thread, while when too many increments are done on the central clock, it reduces the throughput of *all* in-flight transactions, both write transactions and read transactions (as both need to read the clock in `beginTxn()`, line 13).

The second reason is that the restart will occur only if the transactions are non-disjoint, i.e. in the event of a conflict, which means that under a pure 2PL algorithm, they would force at least one of the transactions to restart. In other words, DCTL's design is optimized for *disjoint* workloads, which is ideal as these are the sole workloads in which high scalability *is* attainable. In concurrency controls with serializable isolation [3] (or stronger), workloads where multiple concurrent transactions have read-write or write-write shared data accesses will, by definition, trigger a conflict and force those transactions to be ordered, executing one after the other, thus preventing scalability.

Algorithm 3 DCTL and DCTL-SF implementation

```
68 // DCTL and DCTL-SF algorithm
69 bool isReadConsistent(uint64_t sl) {
70   return (isWUnlocked(sl) && getTS(sl) < tl_readTS) ||
71          getTid(sl) == tl_tid);
72 }
73 bool isWriteConsistent(uint64_t sl){return isReadConsistent(sl);}
```

The majority of disjoint workloads are disjoint because write accesses to data are random, to the point where multiple threads can execute transactions in parallel without having conflicts, neither write-write nor read-write. This randomness implies that such workloads are disjoint in space *and* in time, otherwise they would not be disjoint at all.

There is however an exception to this rule: workloads with thread-specific data, where a thread writes to the same memory location in all, or nearly all consecutive transactions, but no other thread writes or reads from that address, or rarely does so. One example are approximate counters [10, 28] where each sub-counter has been assigned to an individual thread, with this counter being incremented in most transactions but rarely read. Another example is allocator metadata, where each thread can have thread-specific (pool) metadata which it modifies whenever there is an allocation or de-allocation, i.e. modifying the metadata in nearly all write transactions. Such workloads are disjoint in time but not in space, as consecutive transactions by the same thread will modify the same data over and over again.

DCTL handles these workloads with *thread-specific-writes* by not restarting the transaction if the last thread to modify the data was itself (line 70). The number of restarts can be reduced by noting that if the thread-id of the last thread modifying the data is the current thread executing (`tl_tid`) then, by definition, the transaction which made that modification has committed and is visible, which means it is safe to read this data regardless of its timestamp (line 70). The same rule is applied for write-lock acquisition (line 73). The fact that in DCTL the `tid` is kept in the sequence-lock *after*

it has been released, allows for this kind of check. This optimization allows for single-thread execution to *never restart* nor increment the clock, which improves single-threaded performance when compared to previous work. It also brings fourth an extra insight: Even if consecutive transactions are disjoint, on allocation or de-allocation, there are always *conflicts* on writing to the same metadata. In disjoint workloads with allocations or de-allocations, consecutive transactions of the same thread will touch the same allocator metadata and it is therefore imperative that they do not advance the clock, which is the behavior of DCTL and DCOTL, but not GV5 (see Figure 7). Notice that reserving bits in the sequence-lock to store the tid leaves less bits for the timestamp when the number of threads in the system is very large, but this problem can be overcome with the usage of a DCAS operation, where the timestamp utilizes 64 bits and the tid plus extra state of the locks utilizes the adjoining 64 bits.

Two-phase locking algorithms do not need a central clock nor do they need a timestamp per transaction, because when conflicts arise, the locks establish the order between the transactions. This is not the case for concurrency controls relying on optimistic reads, because there is no lock acquired for read-accesses. In DCTL any access to a memory location previously modified by a different thread, triggers a restart and an increase of the clock, while for DCOTL we will reduce even further the increments on the clock.

3.2 DCOTL

Like DCTL, on DCOTL the read-access will cause a restart if the lock is taken by another thread, or if it is above `tl_readTS` but only if it was released last by a different thread. If it's the same thread, then the transaction will proceed. When executing a write-access in DCTL, the transaction will restart if the timestamp on the lock is equal or above `tl_readTS`, even if the lock bit is not set. On DCOTL, the lock is acquired regardless of the timestamp on the lock, restarting only when the lock is taken at that moment in time by another thread, i.e. the `LOCKED` bit is set. It is correct to overwrite data without advancing the global clock because if the previous transaction has released the lock, then that transaction has either committed or restarted, but either way, it is currently releasing its locks or has released all locks. In other words, once a transaction starts to release locks, it will not acquire new locks nor execute new modifications. A read-access must be consistent with previous read-accesses in the same transaction, hence the need to validate the timestamps to ensure all data read was written on a transaction which committed previously to the current transaction starting. It's the read-accesses that need the clock to advance and therefore, it makes sense for read-accesses to be the ones triggering the clock to advance, as needed. At commit time, the central clock is read but never incremented, with solely the restarts triggering an increment (line 36).

Consider the following scenario where DCOTL differs from DCTL, txn 1 reads data element *a* and then txn 2 writes to *a* and to another data element *b*. In DCOTL, txn 1 is allowed to now take the lock and write to *b* however, at the end of the transaction, it will do read-set validation which will fail on *a* because it has been modified by txn2, thus reverting *b* back to the state it was left by txn 2. Notice that all *read-accesses* were done consistently (opaque) and the execution appears to have been done *before* txn2.

Algorithm 4 DCOTL and DCOTL-SF implementation

```
74 // DCOTL and DCOTL-SF algorithm
75 bool isReadConsistent(uint64_t sl) {
76   return (isWUnlocked(sl) && getTS(sl) < tl_readTS) ||
77            (getTid(sl) == tl_tid &&
78             (isWUnlocked(sl) || hasConsistentBit(sl)));
79 }
80 bool isWriteConsistent(uint64_t sl) {
81   return isWUnlocked(sl) || (isWLocked(sl) && getTid(sl) == tl_tid);
82 }
```

One particularity of DCOTL is how to handle read-after-writes in the same transaction. DCOTL can write on a memory location as long as the write-lock is available, taking the write lock before writing however, it can happen that later in the same transaction it will need to *read* from this memory location. If the write operation overwrote the entire data protected by the lock, then the data is consistent however, if the write occurred on a sub-set of the data protected by the sequence lock, and if the timestamp+tid were invalid before the write, then it can be inconsistent to read data from the non-written sub-set, even if the transaction later aborts.

Another example is a scenario where txn 1 reads from location *a*, then txn 2 writes into *a* and later txn 1 overwrites *a*. Such an execution is not serializable, which means txn 1 must restart at read-set validation.

To solve this problem, in our implementation we reserve one extra bit in the sequence lock to indicate if the previous timestamp+tid were valid and set this bit accordingly, during write-lock acquisition, and later cleared at lock release. During the transaction, in `stmRead()`, when `isReadConsistent()` is called, it will check the bit by calling `hasConsistentBit()` if it is write-locked (line 78), thus guaranteeing opacity in all situations.

4 Correctness

We now introduce a sketch of proof for the opacity consistency of DCTL. Opacity [23] is a safety property that captures the intuitive requirements that (1) all operations performed by every *committed* transaction appear as if they happened at some single, indivisible point during the transaction lifetime, henceforth named *opacity point*, (2) no ongoing transaction can see modifications that have not yet been committed by other transactions, and (3) every transaction

observes a *consistent* state of the system, thereby observing all modifications done by previous committed transactions.

A transaction is delimited by an invocation time and a response time. The start of the transaction corresponds to the point in time of its invocation (**tx inv**). The point in time when the transaction's response occurs (**tx res**) corresponds to the commit time of the transaction. For all the examples presented, we show the point in time when the invocation and the response of the transaction occurs, thus defining the time window during which the transaction executes. A restart does *not* imply the end of the transaction and the start of a new transaction, which implies that the response of the transaction has yet to take place.

We define a and b as memory locations which are protected by different seqlocks. Each read access is done through `stmRead()` and each write access is done with `stmWrite()`.

Figure 1 displays the four main scenarios we considered. This is similar to previous work by Guerraoui et al. [22]. The

	Thread 1	Thread 2

Scenario 1	`beginTxn()`
	`  x = stmRead(&a);`
	`  y = stmRead(&b);`
	`endTxn();`
		`beginTxn()`
		`  stmWrite(&a,1);`
		`  stmWrite(&b,1);`
		`endTxn();`

Scenario 2	`beginTxn()`
	`  stmWrite(&a,1);`
	`  t = stmRead(&b);`
	`endTxn();`
		`beginTxn()`
		`  stmWrite(&b,1);`
		`  t = stmRead(&a);`
		`endTxn();`

Scenario 3	`beginTxn()`
	`  t = stmRead(&a);`
	`  stmWrite(&b, 1);`
	`endTxn();`
		`beginTxn()`
		`  t = stmRead(&b);`
		`  stmWrite(&a, 1);`
		`endTxn();`

Scenario 4	`beginTxn()`
	`  x = stmRead(&a);`
	`  y = stmRead(&b);`
	`endTxn();`
		`beginTxn()`
		`  y = stmRead(&b);`
		`  x = stmRead(&a);`
		`endTxn();`

Figure 1. Four fundamental transactional scenarios

first scenario consists of one thread reading a and then b, while another thread writes to a and then b. The second scenario has one thread writing to a and reading b, while another thread writes b and reads a. In the third scenario one thread reads a and writes to b, while another thread reads b and writes to a. The fourth scenario has one thread read a and b, while another thread reads b and then a. There is no scenario where all transactions are composed solely of write accesses, because on those DCTL becomes a pure 2PL.

4.1 First Scenario

In our sketch we start with the scenario shown in Figure 2, where one thread executes a transaction which reads from two distinct variables a and b, while another thread executes a transaction which writes to a and b.

For cases 1) and 2) the execution of transaction 1 (tx1) does not detect any conflict with transaction 2 (tx2). The opacity point of tx1 occurs before the opacity point of tx2, and tx1

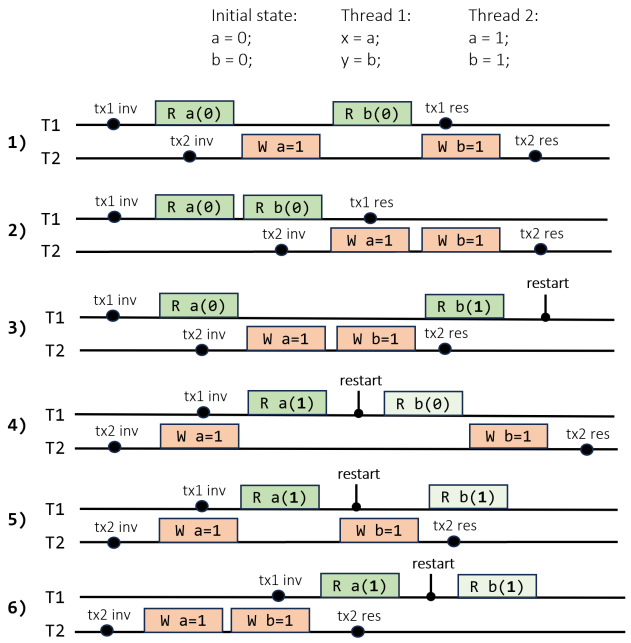

Figure 2. Txn 1 does two reads and txn 2 does two writes

does not see any of the modifications done by tx2. For case 1) the assumption that a and b are protected by different sequence locks, allows the read of b to have no conflict with the previous write of a by tx2. For case 2) would have the same outcome if a and b were protected by the same seqlock.

For case 3) the sequence lock protecting b has been modified by tx2 when tx1 reads b and validates it in `stmRead()`, causing tx1 to restart (leading to an increment of the clock) and re-attempt tx1. The opacity point of tx2 happens-before the opacity point of tx1.

For cases 4) and 5) the sequence lock protecting a has been modified by tx2 when tx1 reads a and validates it in `stmRead()`, causing tx1 to restart and re-attempt tx1. The opacity point of tx2 happens-before the opacity point of tx1.

For case 6) tx2 commits, modifying both a and b without incrementing the clock. This will cause tx1 to fail post-validation after reading a, causing tx1 to restart (leading to an increment of the clock) and re-attempt tx1.

4.2 Second scenario

The second scenario (Figure 3) has one thread writing to a and reading b, while another thread writes b and reads a.

Proof of case 7) is similar to case 4).

For case 8), tx1 will fail the post-validation after reading b because it's locked by tx2, causing tx1 to restart (incrementing the clock) and re-attempt. Even if tx1 releases the sequence-lock of a before tx2 reads it, its timestamp will be equal to the current clock, which is by definition equal or higher than the `readTS` of tx2, therefore invalidating the post-validation after the read of a, thus causing tx2 to restart

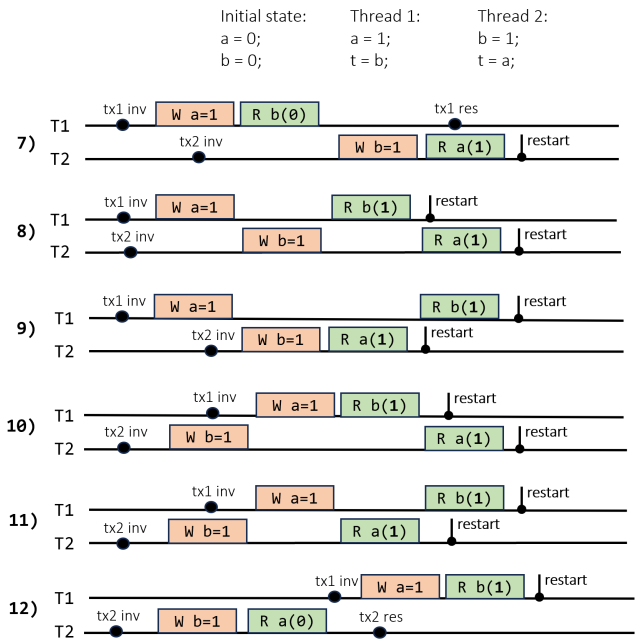

Figure 3. Both transactions do a write followed by a read.

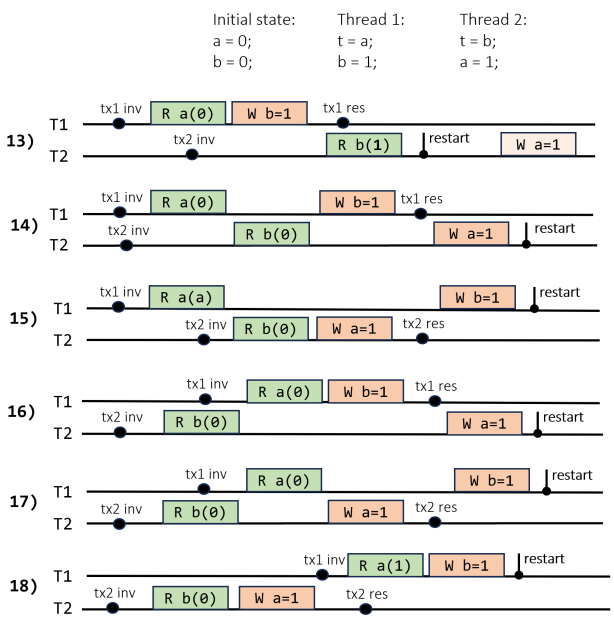

Figure 4. Both transactions do an interdependent read and write.

(incrementing the clock) and re-attempt. This case can cause a live-lock, which is why this algorithm is not starvation-free, an issue which is resolved by the DCTL-SF variant.

Case 9) is similar to case 8) however, in this case the seqlock of a is locked while the seqlock of b may or may not be locked but its timestamp is equal or higher than `readTS`.

Case 10) follows the same logic as case 8).

Case 11) follows the same logic as case 9).

Case 12) is similar to case 7).

4.3 Third scenario

In the third scenario, in Figure 4, one thread reads a and writes to b, while another thread reads b and writes to a.

Case 13) is similar to case 3).

For all remaining cases in this scenario, the read-set validation procedure will fail for one of the transactions, causing it to restart (incrementing the clock) and re-attempt.

4.4 Fourth scenario

The fourth scenario has one thread read a and b, while another thread reads b and then a. There are only two possibilities, either there wasn't any increment on the clock prior to the start of tx1 and tx2, allowing for the seqlock of a or b to be equal to the central clock, and therefore `readTS` of tx1 or tx2. In this case both transactions will restart (incrementing the clock) and re-attempt execution. Otherwise, the timestamps on the seqlocks of a and b are below the central clock, implying that both tx1 and tx2 transactions will commit successfully, without incrementing the clock.

4.5 Summary

In all scenarios, there was never an execution where inconsistent data is returned to the transactional code. As soon as (possibly) inconsistent data is read, the transaction is restarted and all modifications are reverted without ever being observed by the transactional code of other transactions. Also, all effects of previously committed transactions are seen by subsequent transactions. To finish, we argue that any transaction is a linear combination of the previous scenarios, for a finite number of variables [22] and therefore, DCTL provides opacity for generic transactional code.

5 Starvation-Freedom

Starvation-freedom is a vital progress guarantee to ensure fairness and timely execution of transactions (low tail latency). There are several concurrency controls in the literature with starvation-free progress, though few provide opacity and even fewer are able to scale, with examples being PLOR [8] and 2PLSF [43]. These algorithms are pessimistic, being based in Two-Phase Locking (2PL) [4, 17], thus acquiring locks for both read and write accesses to data. These algorithms fall behind the time-based concurrency controls in several workloads due to the higher efficiency of optimistic reads. It is therefore important to devise concurrency controls with optimistic reads that scale well *and* provide starvation-free progress.

With this goal in mind, we devised a general technique to add starvation-free transactions, inspired by the work of Welc et al. [53] and applied this technique to DCTL and

DCOTL, resulting respectively in DCTL-SF and DCOTL-SF. The extra steps to achieve starvation-freedom are shown in blue in Algorithm 1. The SF algorithms differ in the `beginTxn()` and `commitTxn()` functions due to the respective acquisition and release of an *irrevocable lock* (line 1) which grants irrevocable execution. For irrevocable transactions, the `stmRead()` will acquire the read-lock (line 44) and then read the data, while the `stmWrite()` will take the write-lock (line 57). Notice that the speculative transactions follow the non-SF algorithm.

Welc et al. [53] have shown a technique which adds irrevocable transactions to optimistic concurrency controls by having a modified seqlock named SORL, where one of the bits (`RLOCKED`) is reserved for read-locking. A single transaction at a time can execute irrevocably and it will acquire the read-lock on every read data access. Other in-flight transactions can execute concurrent speculative transactions as long as they don't need to write-lock the seqlocks which have been marked with `RLOCKED`. However, Welc's technique is inherently starvable because speculative transactions may indefinitely take and release a SORL lock without ever letting the irrevocable transaction acquire the lock.

We have modified this technique in two significant ways: after K_f failed attempts in speculative mode the transaction will restart in irrevocable mode, *and* our novel reader-writer lock algorithm guarantees starvation-freedom for read-lock and write-lock acquisition (Algorithm 5). When restarting in irrevocable mode, a transaction will wait on the mutual exclusion lock which grants access to execute in irrevocable mode (line 14 of Algorithm 1). For this purpose, our implementation utilizes a ticket-lock [33], but any other mutual exclusion lock with starvation-freedom can be used as its replacement [9, 32, 42]. Once the ticket-lock is granted, the irrevocable transaction may start, acquiring individual write-locks (line 57) for write-accesses, and read-locks (line 44) for read-accesses.

During an irrevocable transaction, a write-lock is acquired by first setting the `RLOCKED` bit with a `fetch_add(RLOCKED)` (line 95), then waiting for the speculative write transaction to depart (line 96). New speculative write transactions will not be able to acquire the write-lock due to the `RLOCKED` bit being set (see line 88), which means that eventually the irrevocable transaction will see the `WLOCKED` bit cleared and be able to set the seqlock to `WLOCKED`, using a store-release (line 97). This technique of using `fetch_add()` differs from the SORL lock proposed by Welc et al. and unlike SORL, ensures starvation-freedom for the lock acquisition.

The lock release in our algorithm also differs from SORL. If the write-unlock procedure were to be done with an atomic store, it could happen that a speculative write-transaction would clear the `WLOCKED` bit at the same time as an irrevocable transaction is setting the `RLOCKED` bit (using `fetch_add(RLOCKED)`), which would also clear the `RLOCKED` bit, thus resulting in incorrect behavior. Even if a retry loop is done, it could theoretically happen that a new speculative write transaction would acquire the lock immediately after, without the irrevocable transaction being able to take the read-lock, which would not provide starvation-free progress. If on the other hand the unlock is done with a `CAS`, it can theoretically fail but it will fail a single time, when another thread running an irrevocable transaction happens to set the `RLOCKED` bit at the same time. Such a transaction will not restart (its execution is irrevocable) and therefore will wait for the speculative write transaction to clear the `WLOCKED` bit, without any further modifications to the seqlock state. In our implementation we execute a `fetch_sub()` operation which simultaneously clears the `WLOCKED` bit and sets the timestamp on the lock to the correct value, thus ensuring no failure during unlock (line 116). For irrevocable transactions, the read-unlock procedure consists of clearing the `RLOCKED` bit with a `fetch_sub(RLOCKED)` (line 112). Figure 5 portrays the bit layout of each sequence-lock for the different algorithms described in this paper.

Algorithm 5 Starvation-free Reader-Writer Sequence lock

```
82 class RWSeqLock {
83   std::atomic<uint64_t> state {0};

85   bool tryWriteLock() { // called from speculative write transactions
86     uint64_t sl = state.load();
87     if (isWLocked(sl)) return (getTid(sl) == tl_tid);
88     if (isRLocked(sl)) return false;
89     return (state.cas(sl, compose(WLOCKED,tl_tid,getTS(sl))));
90   }

92   void writeLock() { // called from irrevocable write transactions
93     uint64_t sl = state.load();
94     if (isWLocked(sl) && getTid(sl) == tl_tid) return true;
95     if (!isRLocked(sl)) state.fetch_add(RLOCKED);
96     while (isWLocked(state.load())) Pause();          // spin loop
97     state.store(compose(WLOCKED,tl_tid,getTS(state.load())));
98   }

100  void readLock() { // called from irrevocable write transactions
101    uint64_t sl = state.load();
102    if (isRLocked(sl)) return;
103    if (isWlocked(sl) && getTid(sl) == tl_tid) return true;
104    state.fetch_add(RLOCKED);
105    while (isWLocked(state.load())) Pause();          // spin loop
106  }

108  void unlock(uint64_t ts=0) { // unlock read/write-locks
109    uint64_t sl = state.load();
110    if (getTid(sl) != tl_tid) return;
111    if (isRLocked(sl) && tl_txType == IRREVOCABLE) {
112      state.fetch_sub(RLOCKED);
113    }
114    if (isWLocked(sl) && getTid(sl) == tl_tid) {
115      uint64_t delta = sl - compose(UNLOCKED, tl_tid, ts);
116      state.fetch_sub(delta);
117    }
118  }
119 };
```

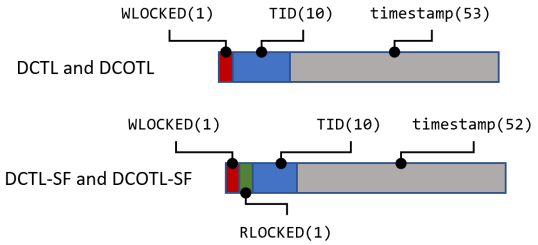

Figure 5. Layout of each sequence-lock (see Algorithm 5)

During an irrevocable transaction, after all the seqlocks have been released, the ticket-lock is released (line 24), allowing another irrevocable transaction to execute. Although only one irrevocable transaction can be executed at a time, there is no bound on the number of speculative write transactions and read transactions simultaneously executing in parallel with the irrevocable transaction. As an added bonus of this technique, both DCTL-SF and DCOTL-SF provide an API for executing irrevocable transactions, an important semantic for applications whose transactional code can not easily support the speculative logic.

Previous work has shown that it is not possible to achieve DAP starvation-free progress with invisible readers [2, 39].

5.1 Sketch of proof of starvation-freedom

Transactions in DCTL-SF and DCOTL-SF will restart at most K_f times, which means they provide starvation-freedom. In `beginTxn()` there is a spin-loop inside the ticket lock's `lock()` call (line 14) however, once the calling thread enters the spin-loop, it will wait at most for $N_t - 1$ threads, with N_t being the total number of threads attempting to execute an irrevocable transaction.

In the calls to `stmWrite()` and `stmRead()` there are spin-loops (lines 96 and 105) which will have to wait for at most one other thread, i.e. a thread which is executing a speculative transaction and has acquired the write-lock on the same seqlock. Once the other thread departs, either because its transaction restarted or committed, no other thread will be able to acquire the lock because it will see the `RLOCKED` bit and return in `tryWriteLock()` line 88, or fail the CAS in line 89. There are no other waiting loops during the execution of a transaction, nor in `commitTxn()`.

The maximum number of times an irrevocable transaction may have to wait for a speculative transaction, is bounded by the number of independent read-accesses plus write-accesses executed during the irrevocable transaction, with each access causing the wait for at most one thread to release the lock. The maximum number of threads an irrevocable transaction may wait for, on the ticket lock, is of $N_t - 1$. This implies DCTL-SF and DCOTL-SF are starvation-free.

6 Durability

TL4x is a Persistent Transactional Memory (PTM) by Assa et al [1] capable of executing scalable opaque speculative write transactions and irrevocable read transactions. TL4x uses TL2 as the underlying concurrency control while persisting to disk with buffered durability [27] using an asynchronous thread running on a consistent snapshot of the DB. The mechanism behind TL4x is effectively a general-purpose replacement for LSM [38] (Log Structured Merge), providing buffered durable transactions while having little impact on the throughput and scalability of the concurrency control.

We have taken TL4x and replaced the TL2 concurrency control with DCTL, giving the name of DCTL4x to the resulting PTM, and did the same for DCOTL. DCTL4x and DCOTL4x provide the same transactional semantics, opacity, and buffered durability as TL4x. The authors of TL4x took their PTM and applied it to a DBMS implementation whose API is equivalent to RocksDB [14, 35] and LevelDB [20], and therefore can run the db_bench benchmarks [35]. Several modifications were needed to give buffered durabliliy to DCTL and DCOTL, which we now succinctly describe:

The asynchronous thread responsible for copying the data from `back` to persistent storage is named the *copy-thread*. Unlike TL4x, neither DCTL nor DCOTL guarantee that the global clock advances at the end of each write transaction, which has implications on the way the copy-thread operates. When a persistent event is started, after changing the state to `PERSIST` and before quiescing with the user-space RCU, the copy-thread must advance the clock and take the current timestamp as the `lastSyncedTS`, by using a `fetch_add()`. It is this timestamp which is placed on the persistent file `P0/P1` at the end of the persistent event. During the persistent event there might be some transactions which will commit and duplicate the corresponding blocks to `back`, therefore updating the corresponding sequence-locks with a later (higher) timestamp than `lastSyncedTS`. The only minor inconvenience is that the next time the same file is written into, on the next-to-next persistent event, those particular blocks will be overwritten even if they have not been modified in the meantime. Notice that `back` is transactionally consistent, it's just the timestamp on the file which may be slightly behind the timestamp on some of the sequence-locks. This is fine because in the event of a power-failure, the central clock is reset to zero and so are the timestamps on the sequence-locks, with the timestamp on the file being used only to determine which of the two persistent files to recover from.

Another modification was to add thread-local slabs to the EsLoco persistent allocator used in the DB. EsLoco is a simplistic allocator which can be used by PTMs, where modifications to allocator metadata are made part of the ongoing transaction.

7 Evaluation

We now present an evaluation of DCTL and DCOTL and compare them with other state-of-the-art STM implementations when applied to transactional data structures, using

synthetic benchmarks. We executed these microbenchmarks on an AMD EPYC 7713 with 64 cores (128 HW threads). This machine was running Ubuntu LTS 20.04 and using `gcc` 10.3.0 with the `-O2` optimization flag.

In the next sections we study the throughput of different transactional map data structures when deployed using an assortment of STM implementations. Half of the keys in the key range (one million) are initially randomly inserted in the map. Each data point represents the mean over 5 runs of 20 seconds. We compared the following STMs:

- TL2-GV4 the original TL2 by Dice et al. [11] with pessimistic writes and optimistic read accesses. Write transactions require read-set validation at commit time;
- TL2-GV5 TL2 with the GV5 optimization from the public repository by Chi Cao Minh [36];
- TinySTM implementation of LSA [18, 19], also with pessimistic writes, optimistic reads, and read-set validation;
- ORec STM implementation by Zardoshti et al [55];
- 2PLSF a pessimistic STM based on two-phase locking with starvation-free transactions, by Ramalhete et al [43];
- TicToc is a serializable (non-opaque) concurrency control by Yu et al. [54] and therefore cannot be used as an STM;

These STMs provide opaque transactions, except for TicToc. We also show four different variants of the DCTL algorithm, two of them being Starvation-Free:

- DCTL / DCTL-SF: Our algorithm where the clock is incremented on restart;
- DCOTL / DCOTL-SF: Optimized DCTL where the clock is incremented only on restarts iif those restarts were triggered from the `stmRead()` function;

7.1 Scalability and atomic increments

Figure 6 displays three plots for operations on a tree with one million key/records, with each operation being a transaction which does a lookup on the key for a random record and modifies a 64bit word on the record with 12 words. All operations are write transactions and they are mostly disjoint. The left plot shows the number of operations, the right plot

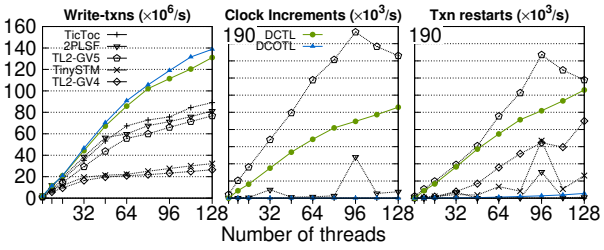

Figure 6. The lower the number of clock increments (center plot) the higher the number of write transactions (left plot).

shows the number of restarts, and the center plot shows the number of atomic increments each STM executes per second, with TinySTM and TL2 incrementing the central clock once per write transaction, while DCTL and DCOTL rarely increment it. We also exercised a workload where threads compete solely to increment a central atomic variable using a `fetch_add()` (not shown in the plots), with the goal of demonstrating the *upper bound* on the total number of such instructions this particular machine can execute per second, specifically, it is lower than 40 million instructions per second for 4 or more threads. This implies that algorithms which require one atomic increment for every write transaction will never be able to go above this plateau of 40 M tps, a behavior which is observable for TinySTM and TL2-GV4 on the left plot of Figure 6. TicToc does not increment a central clock, allowing it to scale to nearly 100 M tps, though still well below what DCTL and DCOTL achieve. 2PLSF increments a central variable only for conflict resolution and therefore, the clock increment is not a bottleneck for this STM instead, the reason for not going above 60 M tps is the lock acquisition for both read and write accesses. On the other hand, DCTL and DCOTL have no such limitation, executing a number of increments per second of respectively 106004 and less than 1. The amount of increments grows linearly with the number of write transactions, implying DCTL and DCOTL are capable of scaling unimpeded, reaching more than 130 million write tps on our machine, and likely continuing to scale on machines with a higher number of cores. Our benchmarks show a minor relative throughput difference between DCTL and DCOTL, which demonstrates that in these algorithms, the clock increments are no longer a scalability bottleneck.

In the allocation-heavy workload of Figure 7, DCTL and DCOTL execute the same number of clock increments because modifications on the RAVL [44] nodes and on the allocator metadata lists have a pattern of read-then-write on the same (and consecutive) transactions, which implies that the DCOTL optimization from `stmWrite()` is rarely done.

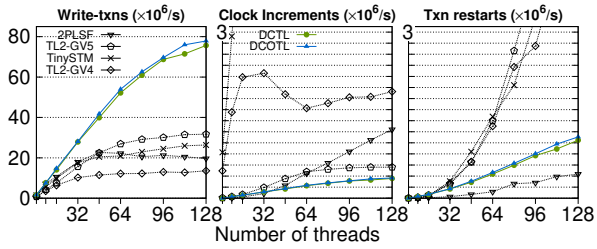

Figure 7. Allocation-heavy workload with 50% insertions and 50% removals on a RAVL.

Transactional restarts (rightmost plot) are high for all STMs except DCTL, DCOTL and 2PLSF. We believe the low number of restarts in 2PLSF is due its starvation-free mechanism causing a thread to wait for the conflicting transactions to complete or restart, instead of restarting itself, thus leading to a low number of restarts but without the corresponding throughput increase. TicToc is not shown in this plot because

it is not opaque and therefore is unable to execute correct mutative operations on most data structures.

Most real-world scenarios will sit somewhere between the two workloads of Figures 6 and 7, executing a few allocations or de-allocations in some, but not all transactions. Nonetheless, Figure 7 highlights the importance of having a concurrency control which is efficient for workloads where the same thread reads and writes to the same locations in consecutive transactions (*thread-specific* data accesses). Although GV5 reduces the number of clock increments, it triggers restarts for thread-specific data patterns, leading to its poor performance in these kind of workloads.

7.2 Starvation-free K_f

Henceforth, for three-plot figures, the leftmost plots are composed of 50% random insertions and 50% removals, the central plots have 5% insertions, 5% removals and 90% lookups, and the rightmost plots executes lookups exclusively.

Figure 8 shows a transactional Rank-based relaxed AVL (RAVL) [25, 44] under the DCOTL an DCOTL-SF STMs with different K_f values, which controls the number of attempts in optimistic mode before falling back to irrevocable pessimistic mode. For write-intensive and mixed workloads (left and

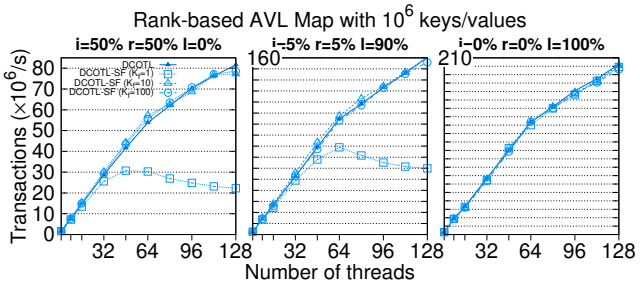

Figure 8. DCOTL-SF with different K_f parameters (1,10,100).

center plots) we notice an important difference between $K_f = 1$ and the other parametrizations, but no improvement from $K_f = 10$ to $K_f = 100$. For read-only workloads (right-side plot), there is no noticeable difference when varying K_f, which is expected because there are no conflicts and therefore no restarts. Results for DCTL-SF are similar to the ones shown in Figure 8 and, for the remaining of the paper, we will use $K_f = 10$ for both DCTL-SF and DCTOL-SF, as being an acceptable trade-off of progress with little sacrifice in throughput.

7.3 Transactional hashmap

Figure 9 shows a transactional data structure, a fixed-size hashmap with 1M bucket entries, wrapped with different STM implementations. This workload is highly disjoint with write transactions being extremely short, exposing the advantage of DCTL and its variants over the other STMs, achieving 250 M (million) write tps (left plot) and 1200 M read tps (right plot). On the leftmost workload, the number of

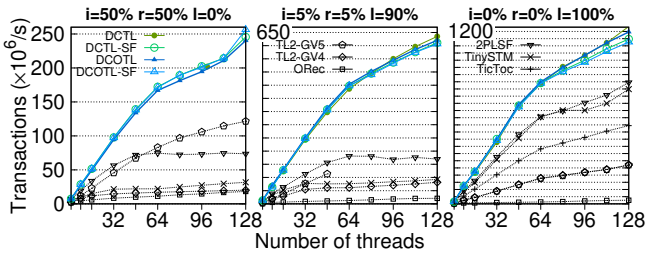

Figure 9. Fixed-size hashmap with one million key/values.

clock increments per second for 128 threads in DCTL and DCOTL is 485518 and 454319, which means that the number of cores and threads could increase by 80x, before the 40 M `fetch_add()` per second becomes a scalability bottleneck. Notice this workload has few write-after-writes on consecutive transactions, causing the clock to advance as much for DCOTL as DCTL. Fixed-size hashmap workloads are rare in the real world however, they show that as long as data accesses are disjoint, the DCTL algorithm can scale far above previous concurrency controls.

7.4 Transactional tree-based maps

In Figure 10 we show four different transactional map data structures: a Rank-based AVL, a ZipTree, a SkipList, and a Red-Black tree. These data structures and workloads are representative of indexing data structures for DBMS.

Figure 11 shows a Rank-based Relaxed AVL [44] for a write-intensive and allocation-intensive workload (left), a mixed workload (center) and a read-only workload (right). Figure 12 we show the write-intensive workloads for a ZipTree [48], a SkipList [40], and a Red-Black tree [24]. The read-only workloads have trends similar to Figure 11.

On a highly scalable data structure like the RAVL or the ZipTree, the difference in scalability between the multiple concurrency controls algorithms is immediately noticeable, particularly on the write-intensive workloads. The Skiplist has poor write scalability beyond 16 threads, achieving less than 1/10 the throughput of the RAVL, and the Red-Black has no scalability at all for write-transactions because each insertion and removal operations will typically trigger a rebalance which will touch the root node, thus causing all other in-flight transactions to restart. On DCTL and its variants, a restart may cause an increment of the clock.

Regarding the STM implementations, for read-mostly workloads DCTL and its variants are always above the other STMs. On the Red-Black tree, DCTL and its variants lag behind TL2-GV4 and TinySTM on the write-intensive workloads with the reason being contention on the root node causing conflicts, which TL2 and TinySTM are better at solving in this particular scenario, as the design of DCTL is optimized for workloads *without* conflicts. For the RAVL and ZipTree, DCTL scales almost linearly, achieving more than 2x the throughput of TinySTM at 128 threads, with the other

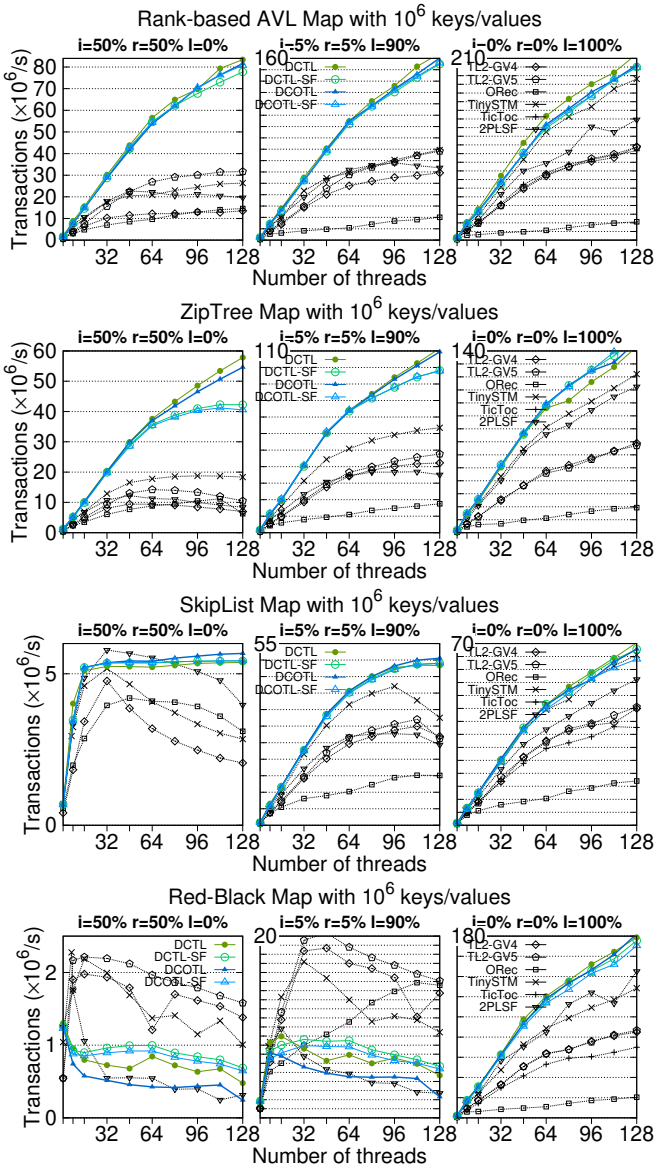

Figure 10. Sequential tree maps protected with STMs. The top row of plots display a RAVL, the second row a ZipTree, the third row a SkipList, and the bottom a Red-Black tree.

STMs staying well below. Due to TL2-GV5's instability, it was not possible to collect measurements for the Skiplist.

These plots provide an important insight regarding concurrent data structures: to obtain scalability we need scalable concurrency controls *and* data structures capable of exploring the full potential of such algorithms.

7.5 Hand-crafted tree-based maps

In Figure 13 we took a sequential map implementation of a RAVL [44] protected with different STMs and compared it with multiple hand-crafted map data structures from the setbench [29] microbenchmark suite:

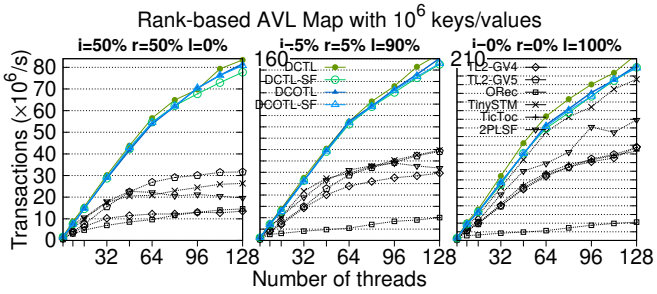

Figure 11. Sequential RAVL tree map protected with STMs.

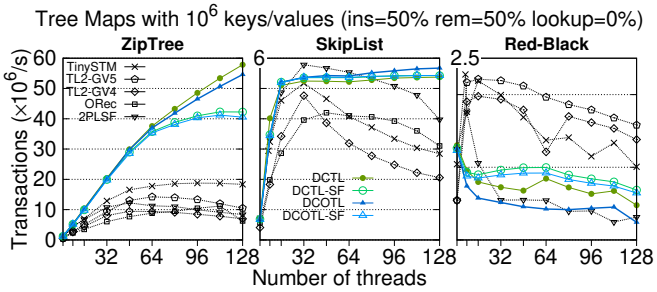

Figure 12. Sequential tree maps protected with STMs.

- howley_int_bst Internal lock-free BST (Binary Search Tree) by Howley and Jones [26];
- ellen_ext_bst External lock-free BST by Ellen et al [15];
- natarajan_ext_bst Non-balanced external lock-free BST by Natarajan and Mittal [37];
- ramachandran_int_bst Internal lock-free BST by Ramachandran and Mittal [41];
- bronson_pext_bst_occ Partially-external AVL tree with locks and optimistic reads, by Bronson et al [5];

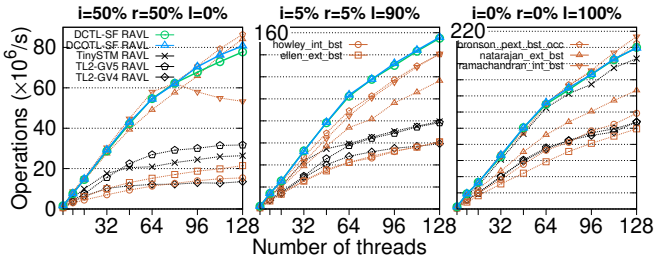

Figure 13. Sequential RAVL protected with STMs versus concurrent hand-crafted map data structures.

Our transactional RAVL provides: average and worst-case logarithmic number of steps for the three main operations (insertion, removal and lookup), range-queries, composable operations, and arbitrary new functionality. A minority of the hand-crafted maps provide logarithmic worst-case steps for their operations, and none of them can provide the other functionalities given by the RAVL+STM. In other words, most of the hand-made are *unbalanced* trees, while the RAVL is a *balanced* tree. The howley and natarajan will become

a linked list if key insertion is in order instead of random. Performance wise, the RAVL+DCTL-SF gets close to the two best hand-made maps on the write-intensive workload (left plot of Figure 13) and matches the best of them for the mixed and read-only workload (center and right plots), while guaranteeing starvation-free progress, which none of the hand-made trees provide.

7.6 Buffered Durable Transactions

We integrated DCTL4x and DCOTL4x with PTM-DB [1], thus creating DCTL4xDB and DCOTL4xDB. As shown in Figure 14, we compared DCTL4xDB and DCOTL4xDB with TL4xDB and RocksDB (8.3.2), all four DBMS with buffered durability enabled and persisting to an SSD device. Both

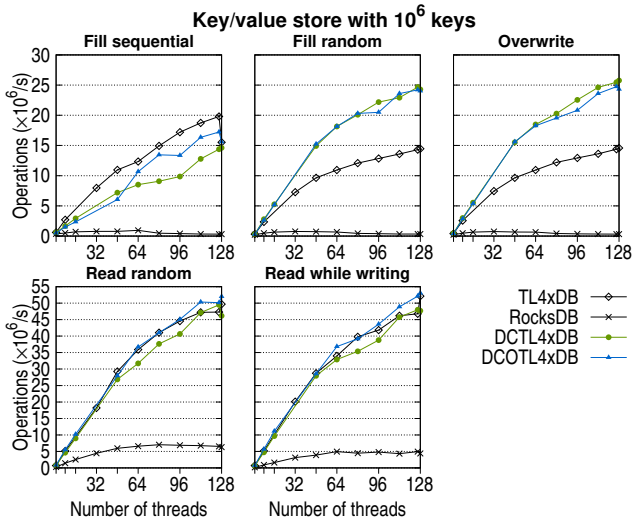

Figure 14. db_bench with 1 million records

DCTL4xDB and DCOTL4xDB are capable of scaling to a high number of cores without write-stalls, with each operation being a transaction.

On `readrandom` at 126 threads, DCTL4xDB matches TL4xDB and surpasses RocksDB by over 7x. On `fillrandom` with the same number of threads, DCTL4xDB surpasses TL4xDB by over 2x and RocksDB by over 70x. In the `fillsequential` benchmark DCTL4xDB has lower results than TL4x because threads compete to insert sequential keys, which causes a high number of conflicts and subsequent restarts. These results demonstrate that when coupled with an efficient durability technique, DCTL can achieve high scalability while providing durability on block storage, making it an ideal concurrency control for implementing transactional in-memory database engines.

8 Discussion

At the time of this writing, the largest node on AWS has 448 vCPUs and 24 TB of RAM [45]. This means that workloads which were previously the exclusive domain of large scale-out systems, can now fit into a single node [34, 50], leading some experts to name this trend as the *death of big data* [49]. Such viewpoints may sound extreme, nonetheless, it is clear that designing concurrency controls which scale up to a high number of cores is becoming increasingly important.

Deferred Clock Transactional Locking (DCTL) is a time-based concurrency control which focuses on improving the scalability of write transactions by reducing the contention on the central clock. For algorithms like TL2 and LSA, the bottleneck generated by the atomic increments on the central clock becomes an issue on scale workloads with short write-transactions. This advantage of DCTL is particularly relevant on DBMS OLTP workloads and concurrent data structures. For DBMS OLAP scenarios and other read-dominated workloads, there is no significant difference in terms of scalability of DCTL versus TL2 and LSA, apart from implementation details. On the other hand, neither TL2 nor LSA provide starvation-freedom, a vital property for workloads where liveness matters. Moreover, starvation-freedom guarantees that every transaction will eventually execute, with a bound on the number of restarts, increasing the overall robustness of the system.

Although we measured no significant performance differences between DCTL and DCOTL, the later executes fewer clock increments in several workloads (like Figure 6), meaning DCOTL is better at eliminating the global clock bottleneck. With its opaque transactions and starvation-freedom, DCOTL-SF provides for the majority of scenarios, at least as good scalability and performance as the previous concurrency control algorithms in the literature, largely surpassing them on short write-intensive workloads.

References

[1] Gal Assa, Andreia Correia, Pedro Ramalhete, Valerio Schiavoni, and Pascal Felber. 2023. TL4x: Buffered Durable Transactions on Disk as Fast as in Memory. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*, Maryam Mehri Dehnavi, Milind Kulkarni, and Sriram Krishnamoorthy (Eds.). ACM, 245–259. https://doi.org/10.1145/3572848.3577495

[2] Hagit Attiya, Eshcar Hillel, and Alessia Milani. 2011. Inherent Limitations on Disjoint-Access Parallel Implementations of Transactional Memory. *Theory Comput. Syst.* 49, 4 (2011), 698–719. https://doi.org/10.1007/S00224-010-9304-5

[3] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, Michael J. Carey and Donovan A. Schneider (Eds.). ACM Press, 1–10. https://doi.org/10.1145/223784.223785

[4] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley. http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx

[5] Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, R. Govindarajan, David A. Padua, and Mary W. Hall (Eds.). ACM,

257–268. https://doi.org/10.1145/1693453.1693488

[6] Irina Calciu, David Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. 2013. NUMA-aware reader-writer locks. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc (Eds.). ACM, 157–166. https://doi.org/10.1145/2442516.2442532

[7] Ved Prakash Chaudhary, Chirag Juyal, Sandeep S. Kulkarni, Sweta Kumari, and Sathya Peri. 2019. Achieving Starvation-Freedom in Multi-version Transactional Memory Systems. In *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11704)*, Mohamed Faouzi Atig and Alexander A. Schwarzmann (Eds.). Springer, 291–310. https://doi.org/10.1007/978-3-030-31277-0_20

[8] Youmin Chen, Xiangyao Yu, Paraschos Koutris, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. 2022. Plor: General Transactions with Predictable, Low Tail Latency. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 19–33. https://doi.org/10.1145/3514221.3517879

[9] Dave Dice and Alex Kogan. 2019. TWA - Ticket Locks Augmented with a Waiting Array. In *Euro-Par 2019: Parallel Processing - 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11725)*, Ramin Yahyapour (Ed.). Springer, 334–345. https://doi.org/10.1007/978-3-030-29400-7_24

[10] Dave Dice, Yossi Lev, and Mark Moir. 2013. Scalable statistics counters. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc (Eds.). ACM, 307–308. https://doi.org/10.1145/2442516.2442558

[11] David Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4167)*, Shlomi Dolev (Ed.). Springer, 194–208. https://doi.org/10.1007/11864219_14

[12] David Dice and Nir Shavit. 2010. TLRW: return of the read-write lock. In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, Friedhelm Meyer auf der Heide and Cynthia A. Phillips (Eds.). ACM, 284–293. https://doi.org/10.1145/1810479.1810531

[13] David Dice, Nir N Shavit, Ori Shalev, and Mark Moir. 2011. Globally incremented variable or clock based methods and apparatus to implement parallel transactions. US Patent 8,028,133.

[14] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf

[15] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking binary search trees. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, Andréa W. Richa and Rachid Guerraoui (Eds.). ACM, 131–140. https://doi.org/10.1145/1835698.1835736

[16] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. 2007. SNZI: scalable NonZero indicators. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, Indranil Gupta and Roger Wattenhofer (Eds.). ACM, 13–22. https://doi.org/10.1145/1281100.1281106

[17] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (1976), 624–633. https://doi.org/10.1145/360363.360369

[18] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. 2010. Time-Based Software Transactional Memory. *IEEE Trans. Parallel Distributed Syst.* 21, 12 (2010), 1793–1807. https://doi.org/10.1109/TPDS.2010.49

[19] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, Siddhartha Chatterjee and Michael L. Scott (Eds.). ACM, 237–246. https://doi.org/10.1145/1345206.1345241

[20] Google. 2023. LevelDB. http://leveldb.org/.

[21] Vincent Gramoli, Rachid Guerraoui, and Vasileios Trigonakis. 2012. TM^{2}C: a software transactional memory for many-cores. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, Pascal Felber, Frank Bellosa, and Herbert Bos (Eds.). ACM, 351–364. https://doi.org/10.1145/2168836.2168872

[22] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. 2010. Model checking transactional memories. *Distributed Comput.* 22, 3 (2010), 129–145. https://doi.org/10.1007/S00446-009-0092-6

[23] Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, Siddhartha Chatterjee and Michael L. Scott (Eds.). ACM, 175–184. https://doi.org/10.1145/1345206.1345233

[24] Leonidas J. Guibas and Robert Sedgewick. 1978. A Dichromatic Framework for Balanced Trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*. IEEE Computer Society, 8–21. https://doi.org/10.1109/SFCS.1978.3

[25] Bernhard Haeupler, Siddhartha Sen, and Robert Endre Tarjan. 2009. Rank-Balanced Trees. In *Algorithms and Data Structures, 11th International Symposium, WADS 2009, Banff, Canada, August 21-23, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5664)*, Frank K. H. A. Dehne, Marina L. Gavrilova, Jörg-Rüdiger Sack, and Csaba D. Tóth (Eds.). Springer, 351–362. https://doi.org/10.1007/978-3-642-03367-4_31

[26] Shane V. Howley and Jeremy Jones. 2012. A non-blocking internal binary search tree. In *24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, Pittsburgh, PA, USA, June 25-27, 2012*, Guy E. Blelloch and Maurice Herlihy (Eds.). ACM, 161–171. https://doi.org/10.1145/2312005.2312036

[27] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9888)*, Cyril Gavoille and David Ilcinkas (Eds.). Springer, 313–327. https://doi.org/10.1007/978-3-662-53426-7_23

[28] Guy L. Steele Jr. and Jean-Baptiste Tristan. 2016. Adding approximate counters. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, Rafael Asenjo and Tim Harris (Eds.). ACM, 15:1–15:12. https://doi.org/10.1145/2851141.2851147

[29] Rosina Kharal and Trevor Brown. 2022. Performance Anomalies in Concurrent Data Structure Microbenchmarks. In *26th International Conference on Principles of Distributed Systems, OPODIS 2022, December 13-15, 2022, Brussels, Belgium (LIPIcs, Vol. 253)*, Eshcar Hillel, Roberto Palmieri, and Etienne Rivière (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:24. https://doi.org/10.4230/

LIPIcs.OPODIS.2022.7

[30] Yossi Lev, Victor Luchangco, and Marek Olszewski. 2009. Scalable reader-writer locks. In *SPAA 2009: Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures, Calgary, Alberta, Canada, August 11-13, 2009*, Friedhelm Meyer auf der Heide and Michael A. Bender (Eds.). ACM, 101–110. https://doi.org/10.1145/1583991.1584020

[31] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 21–35. https://doi.org/10.1145/3035918.3064015

[32] Peter Magnusson, Anders Landin, and Erik Hagersten. 1994. Queue locks on cache coherent multiprocessors. In *Proceedings of 8th International Parallel Processing Symposium*. IEEE, 165–171.

[33] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (1991), 21–65. https://doi.org/10.1145/103727.103729

[34] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* 3, 1 (2010), 330–339. https://doi.org/10.14778/1920841.1920886

[35] Meta. 2017. RocksDB. http://rocksdb.org/.

[36] Chi Cao Minh. 2023. TL2-x86. https://github.com/ccaominh/tl2-x86/blob/master/tl2.c.

[37] Aravind Natarajan and Neeraj Mittal. 2014. Fast concurrent lock-free binary search trees. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*. 317–328. https://doi.org/10.1145/2555243.2555256

[38] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

[39] Sebastiano Peluso, Roberto Palmieri, Paolo Romano, Binoy Ravindran, and Francesco Quaglia. 2015. Disjoint-Access Parallelism: Impossibility, Possibility, and Cost of Transactional Memory Implementations. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, Chryssis Georgiou and Paul G. Spirakis (Eds.). ACM, 217–226. https://doi.org/10.1145/2767386.2767438

[40] William W. Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (1990), 668–676. https://doi.org/10.1145/78973.78977

[41] Arunmoezhi Ramachandran and Neeraj Mittal. 2015. A Fast Lock-Free Internal Binary Search Tree. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4-7, 2015*, Sajal K. Das, Dilip Krishnaswamy, Santonu Karkar, Amos Korman, Mohan J. Kumar, Marius Portmann, and Srikanth Sastry (Eds.). ACM, 37:1–37:10. https://doi.org/10.1145/2684464.2684472

[42] Pedro Ramalhete and Andreia Correia. 2016. Tidex: a mutual exclusion lock. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, Rafael Asenjo and Tim Harris (Eds.). ACM, 52:1–52:2. https://doi.org/10.1145/2851141.2851171

[43] Pedro Ramalhete, Andreia Correia, and Pascal Felber. 2023. 2PLSF: Two-Phase Locking with Starvation-Freedom. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*, Maryam Mehri Dehnavi, Milind Kulkarni, and Sriram Krishnamoorthy (Eds.). ACM, 39–51. https://doi.org/10.1145/3572848.3577433

[44] Siddhartha Sen, Robert E. Tarjan, and David Hong Kyun Kim. 2016. Deletion Without Rebalancing in Binary Search Trees. *ACM Trans. Algorithms* 12, 4 (2016), 57:1–57:31. https://doi.org/10.1145/2903142

[45] Amazon Web Services. 2023. Amazon Nitro System - EC2 High Memory Instances. https://aws.amazon.com/ec2/instance-types/high-memory/.

[46] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. 2009. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*, Daniel A. Reed and Vivek Sarkar (Eds.). ACM, 141–150. https://doi.org/10.1145/1504176.1504199

[47] Takayuki Tanabe, Takashi Hoshino, Hideyuki Kawashima, and Osamu Tatebe. 2020. An Analysis of Concurrency Control Protocols for In-Memory Databases with CCBench (Extended Version). *CoRR* abs/2009.11558 (2020). arXiv:2009.11558 https://arxiv.org/abs/2009.11558

[48] Robert E. Tarjan, Caleb C. Levy, and Stephen Timmel. 2019. Zip Trees. In *Algorithms and Data Structures - 16th International Symposium, WADS 2019, Edmonton, AB, Canada, August 5-7, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11646)*, Zachary Friggstad, Jörg-Rüdiger Sack, and Mohammad R. Salavatipour (Eds.). Springer, 566–577. https://doi.org/10.1007/978-3-030-24766-9_41

[49] Jordan Tigani. 2023. Big Data is Dead. https://motherduck.com/blog/big-data-is-dead/.

[50] Jordan Tigani. 2023. The Simple Joys of Scaling Up. https://motherduck.com/blog/the-simple-joys-of-scaling-up/.

[51] M. M. Waliullah and Per Stenström. 2009. Schemes for avoiding starvation in transactional memory systems. *Concurr. Comput. Pract. Exp.* 21, 7 (2009), 859–873. https://doi.org/10.1002/cpe.1363

[52] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *Proc. VLDB Endow.* 10, 2 (2016), 49–60. https://doi.org/10.14778/3015274.3015276

[53] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. 2008. Irrevocable transactions and their applications. In *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 14-16, 2008*, Friedhelm Meyer auf der Heide and Nir Shavit (Eds.). ACM, 285–296. https://doi.org/10.1145/1378533.1378584

[54] Xiangyao Yu, Andrew Pavlo, Daniel Sánchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1629–1642. https://doi.org/10.1145/2882903.2882935

[55] Pantea Zardoshti, Tingzhe Zhou, Yujie Liu, and Michael F. Spear. 2019. Optimizing Persistent Memory Transactions. In *28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019*. IEEE, 219–231. https://doi.org/10.1109/PACT.2019.00025