2PLSF: Two-Phase Locking with Starvation-Freedom

Pedro Ramalhete
Cisco Systems
pramalhe@gmail.com

Andreia Correia
University of Neuchâtel
andreia.veiga@unine.ch

Pascal Felber
University of Neuchâtel
pascal.felber@unine.ch

Abstract

Invented more than 40 years ago, the *two-phase locking* concurrency control (2PL) is capable of providing opaque transactions over multiple records. However, classic 2PL can suffer from live-lock and its scalability is low when applied to workloads with a high number of non-disjoint read accesses.

In this paper we propose a new 2PL algorithm (2PLSF) which, by utilizing a novel reader-writer lock, provides highly scalable transactions with starvation-freedom guarantees. Our 2PLSF concurrency control can be applied to records, metadata and to indexing data structures used in database management systems (DBMS). In our experiments we show that 2PLSF is often superior to classical 2PL and can surpass concurrency controls with optimistic reads, simultaneously providing high throughput and low latency for disjoint and non-disjoint workloads.

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

Keywords Concurrency control, transactions

1 Introduction

Invented back in 1976 in the context of database management systems (DBMS), two-phase locking (2PL) was the first of the general purpose concurrency controls to provide serializable transactions [1, 10]. In 2PL, before accessing a record, whether for reading or writing, the lock that protects it must be acquired. At the end of the transaction the locks are released. The name *two-phase* comes from the two distinct phases: lock acquisition is done during the first (expanding) phase and on the second (shrinking) phase the locks are released. To guarantee serializability, once the first lock is released, no further locks can be acquired.

More specifically, 2PL ensures *opacity* [14], a condition stronger than serializability. Opacity guarantees that no ongoing transaction can see modifications that have not yet been committed. In practice this implies that all transactions are serializable, for both committed as well as aborted transactions. This behavior simplifies programming by ensuring that application invariants hold during transaction execution [23].

2PL implementations typically use a mutual exclusion lock even for read accesses [29], the reason being that common reader-writer locks have poor scalability for short lived read-lock acquisitions. This happens because of contention on the variable that counts the number of active readers, henceforth named the *read-indicator*. A read-indicator [9, 17] is a concurrent object that provides three functions: `arrive()`, `depart()` and `isEmpty()`. While there exists in the literature scalable reader-writer locks that address reader scalability by splitting the read-indicator over multiple cache lines, so as to reduce contention [2, 17], we know of no 2PL implementations that utilize these reader-writer locks.

As a result, existing 2PL implementations suffer from poor scalability for read-mostly workloads when all or most transactions read the same objects. This is particularly noticeable when using 2PL as the concurrency control of tree-based indexing data structures, where all threads must read the root node of the tree to access their intended key. In other words, all operations on the tree will need to obtain the read-lock protecting the root node, causing a scalability bottleneck. Hence the reason for DBMS preferring the usage of indexing data structures whose concurrency control have optimistic reads. Optimistic concurrency control techniques execute read accesses without acquiring read-locks and instead use *sequence locks* or similar versioning mechanisms to check if the data was modified during the read access.

When it comes to the way conflicts are handled, there are three main variants of 2PL, named *no-wait*, *wait-or-die* and *deadlock-detection* [1, 28, 29]. The 2PL *no-wait* variant uses a backoff strategy, aborting the transaction as a soon as a lock conflict is detected. The 2PL *wait-or-die* variant imposes an order on all transactions, typically with a timestamp of when the transaction started and, when a lock conflict arises, decides to wait for the lock or to abort, by comparing the timestamp of the transaction with the timestamp of the lock owner. The 2PL *deadlock-detection* variant keeps an internal list of threads waiting on a lock and detects cycles (deadlocks). A variant of wait-or-die exists named *wound-wait* in which the transactions causing the conflict are aborted [4], as opposed to wait-or-die where the transaction aborts itself. The *wait-or-die* and *deadlock-detection* approaches can be implemented with live-lock freedom. To the best of our knowledge, at the exception of PLOR [4] no other algorithm based on these variants has been proposed with starvation-freedom, a vital progress condition for guaranteeing the timely execution of each transaction.

In this paper we propose 2PLSF, a two-phase locking concurrency control with a conflict strategy similar to 2PL *wait-or-die* however, 2PLSF is capable of providing scalable

This is the author's version of the work. It is posted here for your personal use. The definitive Version of Record was published in PPoPP '23 http://dx.doi.org/10.1145/3572848.3577433,
2023.

starvation-free transactions. Our goal is to show that 2PLSF can provide scalability for DBMS, including indexing data structures, and can therefore be simultaneously deployed in these data structures and on the records of the database, thus simplifying the implementation and maintenance of DBMS. 2PLSF achieves this via two distinctive characteristics.

The first characteristic is the usage of a highly scalable reader-writer lock which splits the read-indicator over multiple cache lines, with one bit per thread per read-indicator, aggregating into a single 64 bit-word the read-indicators of 64 reader-writer locks (for one thread). This approach reduces memory usage and allows read-lock releases to be made with an atomic store-release, an operation with a lower synchronization cost than a sequentially-consistent store or an atomic decrement.

The second characteristic is the usage of a centralized atomic counter to order the transactions only when conflicts arise. Unlike 2PL *wait-or-die* which increments a central clock for *every* transaction, the reader-writer locks in 2PLSF increment the central clock at most *one* time per transaction, *i.e.*, the first time a conflict is detected, causing less contention on the central clock variable, which results in higher scalability. Moreover, our reader-writer lock allows 2PLSF to keep track of which thread/transaction caused the conflict, implying that the aborted transaction needs only to wait for the conflicting transaction, whilst in 2PL *wait-or-die* the aborted transaction must wait for *all* transactions with a lower timestamp, even if they are non-conflicting. This results in an efficient conflict resolution for 2PLSF, particularly in workloads with pair-wise conflicts, where it is capable of providing scalability.

Summarizing, the main contributions of this paper are:

- a new reader-writer lock which provides starvation-freedom and due to its unique memory layout allows for high scalability on read accesses;
- a novel two-phase locking algorithm (2PLSF) which, by utilizing the aforementioned reader-writer lock, provides starvation-free and opaque transactions that are guaranteed to restart at most $N_{threads} - 1$ times;
- in 2PLSF only conflicting transactions are ordered and each conflicting transaction needs to wait only for the specific transaction(s) that caused the conflict.

The rest of the paper is organized as follows. We first present our concurrency control 2PLSF in §2. We perform an evaluation of 2PLSF in §3. We then discuss related work in §4. Finally we conclude in §5.

2 Starvation-Free Two-Phase Locking

Concurrency controls like 2PL can manage accesses to data at the individual read and write accesses. When deployed in a *software transactional memory* (STM), these are implemented as the read and write interposing functions, namely `stmRead()` and `stmWrite()`, as shown in Algorithm 1.

Algorithm 1 — STM functions

```
1  // total number of reader–writer locks
2  uint64_t NUM_RWL = 4 * 1024 * 1024ULL;        // 4 million locks
3  // timestamp array with all entries initialized to NO_TIMESTAMP
4  atomic<uint64_t> announceTS[MAX_THR];
5  // default value in announceTS[]
6  uint64_t NO_TIMESTAMP = 0;
7  // central clock for conflicts
8  atomic<uint64_t> conflictClock = 1;
9  // unique thread id for a thread
10 thread_local uint16_t tl_tid;

11 T stmRead(T* addr) {
12   if (!tryOrWaitReadLock(addr2lockIdx(addr))) restartTxn();
13   readlockSet.log(addr);              // save address to later unlock
14   return *addr;                                          // read data
15 }

16 void stmWrite(T* addr, T newValue) {
17   if (!tryOrWaitWriteLock(addr2lockIdx(addr))) restartTxn();
18   writeSet.log(addr, *addr);      // save address and original value
19   *addr = newValue;                                     // write to data
20 }

21 void beginTxn() {          // always inlined or preprocessor macro
22   setjmp();
23   readSet.reset();
24   writeSet.reset();
25   // will do nothing on the first attempt
26   while (announceTS[tl_otid] == tl_oTS) pause();
27 }

28 void commitTxn() {
29   writeSet.unlock();
30   readSet.unlock();
31   tl_myTS = NO_TIMESTAMP;
32   announceTS[tid].store(NO_TIMESTAMP);
33 }

34 void restartTxn() {
35   writeSet.rollbackModifications();
36   writeSet.unlock();          // calls writeUnlock() for every entry
37   readSet.unlock();           // calls readUnlock() for every entry
38   longjmp();                  // continues at setjmp() in beginTxn()
39 }

40 // returns the index of the lock which protects a given address
41 uint32_t addr2lockIdx(void* addr) {    // example of lock hashing
42   return (((uint64_t)(addr) >> 5) & (NUM_RWL-1));
43 }
```

Just like 2PL [1], a lock must be taken before doing a read or write access. If the lock acquisition fails (in lines 12 or 17) the transaction is restarted, which will revert any writes done so far in the transaction and release the locks (line 34) followed by jumping back to the beginning of the transaction in `beginTxn()`. Our implementation uses a *write-through* protocol (undo-log) nevertheless, a *write-back* protocol (redo-log) can also be used with either eager locking or deferred locking [12].

Algorithm 2 — Reader-writer lock used by 2PLSF

```
44 atomic<uint16_t> wlocks[NUM_RWL];          // initialized to zero
45 // thread identifier of the thread causing conflict with this one
46 thread_local uint16_t tl_otid = NO_TID;
47 // timestamp/priority of the thread causing conflict (tl_otid)
48 thread_local uint64_t tl_oTS = NO_TIMESTAMP;
49 // timestamp/priotiy of the current thread's transaction
50 thread_local uint64_t tl_myTS = NO_TIMESTAMP;

51 bool tryOrWaitReadLock(uint32_t widx) {
52   riArrive(widx);
53   uint16_t wstate = wlocks[widx].load();
54   if (wstate == UNLOCKED || wstate == tl_tid) return true;
55   if (tl_myTS == NO_TIMESTAMP) {
56     tl_myTS = conflictClock.fetch_add(1);
57     announceTS[tl_tid].store(tl_myTS);
58   }
59   while (true) {
60     if (wlocks[widx].load() == UNLOCKED) return true;
61     tl_oTS = getTSOfWLock(widx);
62     if (tl_oTS < tl_myTS) {
63       // write-lock taken by other thread with lower timestamp
64       riDepart(widx);
65       return false;
66     }
67     pause();                                   // wait for now
68   }
69 }

70 void readUnlock(uint32_t widx) {
71   riDepart(widx);
72 }

73 void writeUnlock(uint32_t widx) {
74   wlocks[widx].store(UNLOCKED);
75 }

76 bool tryOrWaitWriteLock(uint32_t widx) {
77   uint16_t wstate = wlocks[widx].load();
78   if (wstate == UNLOCKED && wlocks[widx].cas(wstate,tl_tid)) {
79     if (ri.isEmpty(widx)) return true;
80   }
81   if (tl_myTS == NO_TIMESTAMP) {
82     tl_myTS = conflictClock.fetch_add(1);
83     announceTS[tl_tid].store(tl_myTS);
84   }
85   riArrive(widx);                          // writer arrives as a reader
86   while (true) {
87     wstate = wlocks[widx].load();
88     if (wstate == UNLOCKED) wlocks[widx].cas(wstate, tl_tid);
89     wstate = wlocks[widx].load();
90     if (wstate == tl_tid && ri.isEmpty(widx)) {
91       // ok to clear the read-indicator even if it was previously
92       // read-locked because the lock is now upgraded
93       riDepart(widx);
94       return true;
95     }
96     tl_oTS = getLowestTS(widx);
97     if (tl_oTS < tl_myTS) {
98       // write-/read-lock taken by thread with lower timestamp
99       riDepart(widx);
100      if (wlocks[widx].load() == tl_tid)
101        wlocks[widx].store(UNLOCKED);
102      return false;
103    }
104    pause();                                  // wait for now
105  }
106 }
```

2.1 Algorithm overview

Transactions in 2PLSF acquire a read-lock before read accesses and a write-lock before write accesses. The first time a lock conflict occurs during a transaction, a timestamp is taken from a global atomic clock named `conflictClock` and this number is announced in an array with one entry per thread, `announceTS[tl_tid]` (lines 57 and 83). When a conflicting lock is found, the thread failing to acquire the lock (the conflicted transaction) will read the thread-id (`tid`) of the thread currently holding the lock and read the corresponding entry in `announceTS[tid]` to determine the timestamp of the conflicting transaction. In case of conflict when there are multiple threads holding a read-lock, the conflicted transaction will scan the timestamps of the transactions holding the read-locks and wait for the the transaction with the lowest of the timestamps (line 125). If the timestamp of the transaction holding the lock is higher than the timestamp of the transaction attempting to acquire the lock, then the conflicted transaction waits for the lock to be released, otherwise the conflicted transaction will undo its modifications, release all the locks and restart (line 34). The conflicted transaction will not begin again until the transaction which caused the conflict completes (line 26).

When the other transaction commits, it will clear its announced timestamp (in line 32) allowing waiting thread(s) to start and re-attempt their transaction(s). This behavior is possible because each thread that attempts a read-lock will announce its arrival separately on the read-indicator, whilst the majority of reader-writer locks use atomic counters [2, 17]. Our reader-writer lock uses this mechanism to identify which thread has caused the conflict (lines 65 and 102) saving this information in a thread-local variable (`tl_otid`) along with the corresponding timestamp (`tl_oTS`).

Consider the difference from this behavior to 2PL *no-wait* where each aborted transaction waits for a backoff period and therefore is prone to live-lock issues, or the difference from 2PL *wait-or-die* which must wait for *all* other threads whose timestamp is lower, even if they are not in conflict.

2.2 Starvation-Freedom

In the context of STMs the property of starvation-freedom is sometimes defined as: every aborted transaction that is retried infinitely often eventually commits [3]. Our 2PLSF

algorithm provides a stronger guarantee, with a bound on the number of times each transaction will be retried.

When a read-write conflict occurs, the thread detecting the conflict will enter the slow-path in `tryOrWaitReadLock()` and, if it is the first restart, will take a timestamp (line 56) to determine its priority relative to the other transactions. It will then wait (line 67) until the read-lock is successfully acquired (line 60), or a transaction with a lower timestamp is currently holding the write-lock (line 65) in which case it will restart its transaction, reverting all modifications and releasing all locks acquired so far and then wait for the conflicting transaction to commit before restarting the transaction (line 26).

When a write-write or write-read conflict occurs, the thread detecting the conflict will enter the slow-path in `tryOrWaitWriteLock()` and, if it is the first restart, will take a timestamp (line 82) to determine its priority relative to the other transactions. It will then wait (line 104) until the write-lock is successfully acquired (line 94), or a transaction with a lower timestamp is currently holding the write-lock or the read-lock (line 96) in which case it will restart its transaction, reverting all modifications and releasing the locks acquired so far and then wait for the conflicting transaction to commit before restarting the transaction (line 26).

A transaction may restart due to a write-write, read-write or write-read conflict, however the number of restarts is bounded by $N_{threads} - 1$. Once the timestamp is announced in `announceTS[tid]` (line 57 or 83) it will not be cleared until the transaction commits (line 32). All later conflicting transactions will, by definition, have a higher timestamp, taken from `conflictClock`. This means the current transaction has to restart *at most* $N_{threads} - 1$ times due to other transactions before it commits because, after $N_{threads} - 1$ attempts, the transaction will become the in-flight transaction with the lowest timestamp and therefore, in the event of a subsequent lock conflict, it will *wait* for the current owner thread to release the lock. Notice that the other thread will release the lock either because it committed or because it detected a conflict with a transaction with a lower timestamp.

2.3 Why starvation-free locks are not enough

Several mutual exclusion locks with starvation-free progress exist in the literature [19, 21, 25] and reader-writer locks with starvation-freedom. Although we know of no highly scalable reader-writer lock with starvation-free progress for both read and writer lock acquisitions, even if such an algorithm were to be used, it would not suffice to obtain a starvation-free concurrency control.

To understand why, consider a scenario with two transactions, with one of them accessing record A followed by record B, while the other transaction accesses record B followed by record A. Assuming each record is protected by an individual lock, if the lock of A is acquired by the first transaction and the lock of B is acquired by the second transaction, then both transactions are unable to access the opposite record, creating a deadlock situation. This will occur even if each of the locks provide starvation-free progress.

Notice that mutual exclusion lock algorithms with starvation-freedom, do so through the `lock()` API however, concurrency controls acquire locks using the `trylock()` API, which by definition is not starvation-free, due to `trylock()` implying no waiting. As such, our 2PLSF algorithm uses a different API we named `tryOrWaitLock()`, which may wait for a lock acquisition, or return true on successful lock acquisition, or return false when the lock acquisition fails.

2.4 Reader-Writer Lock

Our reader-writer lock was designed with two main premises in mind: resolve conflicts only when they occur and lay out the read-indicator to reduce false-sharing.

The first premise is that there should be no need to order threads/transactions unless there is a conflict. When deployed in the context of a concurrency control, the majority of lock acquisition attempts are expected to be successful, *i.e.*, no lock conflict occurs. As such, we optimize our algorithm by having a short code path for the successful acquisition, in `tryOrWaitReadLock()` lines 52–54 and in `tryOrWaitWriteLock()` lines 76–79. It is only when this *fast-path* fails to acquire the lock that it takes a timestamp from the `conflictCLock` so as to establish a precise order of execution for the threads/transactions (lines 56 and 82). This behavior may seem to differ slightly from ordering *every* transaction, which is what 2PL *wait-or-die* does, however, by incrementing the `conflictCLock` only when needed, it removes a significant bottleneck that prevents the scalability of 2PL *wait-or-die*. We will show just how large these effects can be in section 3.

The second premise is that by using a customized read-indicator, 2PLSF is able to reduce the memory usage to one bit per thread and per lock while preventing false sharing in the read-indicator. Figure 1 shows the memory layout of the scalable rw-locks used in our implementations. Our reader-writer lock is laid out such that each thread has one bit reserved for the read-indicator for each lock.

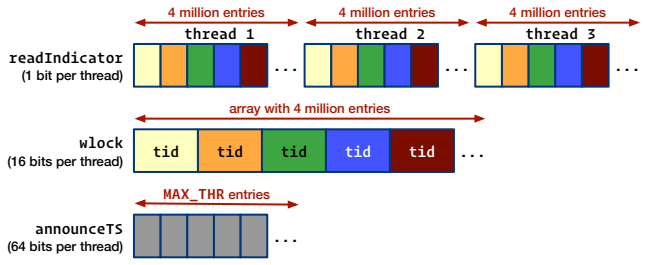

Figure 1. Memory layout of the reader-writer locks used in 2PLSF. `wlock` requires at most 16 bits to store a unique thread identifier, thus supporting 2^{16} threads. Blocks of the same color represent the variables of one reader-writer lock.

Starvation-free locks typically use a atomic increment instruction (`fetch_add()`) to count waiting threads. In our reader-writer lock the bits are consecutive for each thread, implying that setting a read-indicator back to zero can be made with a `store(memory_order_release)` instead of a `fetch_add()`, which is a faster operation in most architectures, including x86. In this design, the memory usage per lock is one bit per number of threads plus the number of bits required to store a thread id (16 bits in our implementation). Moreover the usage of a separate bit in the read-indicator for each thread is vital, so that conflicting threads can determine *which* thread is currently holding the read-lock or waiting for the write-lock. In our implementation we used 4 million reader-writer locks (line 96) with each lock protecting 32 bytes of data (line 41), but other settings can be chosen.

In Algorithm 3 we show our read-indicator implementation and the auxiliary functions that find the lowest timestamp at each conflict.

2.5 RW-Lock TryOrWaitLock

When a conflict occurs in `tryOrWaitReadLock()` or `tryOrWaitWriteLock()`, a unique timestamp is assigned to this transaction if none has yet been assigned, *i.e.*, if this is the first time a lock conflict has been encountered in this transaction. The timestamp is then announced in a thread-specific entry of an announcement array `announceTS[tl_tid]` (line 57). The `tryOrWaitReadLock()` function then enters a waiting loop from which it exits only if one of the following conditions occurs:

- The transaction holding the write-lock has announced a lower timestamp: restart the current transaction by returning false (line 65).
- The transaction holding the write-lock has released it: this transaction has now acquired the read-lock. Continue executing the current transaction by returning true (line 60).

The `tryOrWaitWriteLock()` function enters a waiting loop from which it exits only if one of the following conditions occurs.

- The transaction holding the write-lock has announced a lower timestamp: restart the current transaction by returning false (line 102).
- The transaction in the read-indicator whose announced timestamp is the lowest has a lower timestamp than the current transaction: restart the current transaction by returning false (line 102).
- The write-lock has been acquired with a CAS on `wstate` (line 78 or 88) and the read-indicator is empty (line 79 or 90): return true to continue executing the current transaction (line 94).

When a thread calls `tryOrWaitWriteLock()` for a lock, if there is another thread currently holding the write-lock or multiple threads holding the read-lock, as long as they have a timestamp higher than the thread's timestamp, the calling thread will wait until the lock is released.

One important detail is that when thread i calls `tryOrWaitWriteLock()` and there is another thread j holding the write-lock with a higher timestamp, thread i will mark its read indicator for this lock (line 85). Arriving on the read-indicator ensures that, even if the other thread j (with a higher timestamp) does a successful CAS on `wstate` (line 88), thread j will not see an empty read-indicator after the CAS (line 90) and will restart when it sees a lower timestamp (line 96). Thus, by having the writer mark the read-indicator, we guarantee the number of writers that can take the lock (with a higher timestamp) is bounded.

2.6 Correctness

Similarly to classic 2PL, in 2PLSF all data accesses are done under the protection of a lock. In `stmRead()` we first acquire the read-lock (line 12) and then read the data (line 14). In `stmWrite()` we first acquire the write-lock (line 17) and then write to the memory location (line 19). All locks are released on restart (lines 36 and 37) or on commit of the transaction (lines 29 and 30), thus guaranteeing opacity like classic 2PL.

All that remains is showing that the reader-writer lock used by 2PLSF provides correct mutual exclusion. This can be split into three mutual exclusion scenarios: write-write, read-write and write-read. Write-write mutual exclusion is guaranteed through the compare-and-swap instruction which atomically changes `wlocks[widx]` from `UNLOCKED` to the thread's unique identifier, in line 78 or line 88, thus ensuring that only one thread a time will have the write-lock. Write-read mutual exclusion is provided by checking that the read-indicator is empty after setting the writer's state in `wlocks[widx]`, in lines 79 and 90. This means that a writer will not have the lock when a reader is already in the read-critical section. Read-write mutual exclusion is achieved by checking the writer's state after setting the read-indicator, in lines 53 and 60. This implies that a reader will not have the lock when a writer is already in the write-critical section. This sketch of proof shows that our read-writer lock provides correct mutual exclusion.

2.7 Debuggability

On a practical note, 2PL algorithms have an important advantage over concurrency controls with optimistic accesses: when using a debugger to inspect variables in a transaction, for example in the case of a error introduced by the user in the transactional code, for STMs with optimistic access any variable previously read during the transaction may in the meantime have been modified by another thread's transaction. 2PL and 2PLSF have no such limitation because they acquire locks for every read and write access to the data, thus providing a consistent view of all the variables read up until the point in time where the transaction was stopped. This means that it is significantly easier to utilize tools such

Algorithm 3 — Auxiliary functions and variables for the reader-writer locks

```
// 64bit words used per thread on the read-indicator
uint64_t NUM_RI_WRD = NUM_RWL * MAX_THR / 64;

// initialized to zero
atomic<uint64_t> readIndicators[NUM_RI_WRD][MAX_THR];

// sets the bit on the read-indicator for an rw-lock/tid
void riArrive(uint32_t widx) {
    uint32_t ridx = widx2ridx(widx);
    uint64_t ri = readIndicators[ridx].load();
    readIndicators[ridx].store(ri | ribit(widx));
}

// clears the bit on the read-indicator for an rw-lock/tid
void riDepart(uint32_t widx) {
    uint32_t ridx = widx2ridx(widx);
    uint64_t ri = readIndicators[ridx].load();
    readIndicators[ridx].store(ri & (~ribit(widx)));
}

// returns true if the read-indicator has no active readers
bool riIsEmpty(uint32_t widx) {
    for (uint16_t itid = 0; itid < maxThreads; itid++) {
        if (itid == tid) continue;
        uint64_t offset = itid * NUM_RI_WRD/MAX_THR;
        uint64_t ri = readIndicators[offset + (widx / 64)].load();
        if (ri & ribit(widx)) return false;
    }
    return true;
}

uint32_t widx2ridx(uint32_t widx) {
    return tid * NUM_RI_WRD / MAX_THR + (widx / 64);
}

// returns the lowest timestamp of any conflicting thread
uint64_t getLowestTS(uint32_t widx) {
    uint64_t lowestTS = getTSOfWLock(widx);
    for (uint16_t itid = 0; itid < maxThreads; itid++) {
        if (itid == tid) continue;
        uint64_t offset = itid*NUM_RI_WRD/MAX_THR;
        uint64_t ri = readIndicators[offset + (widx / 64)].load();
        if ((ri & ribit(widx)) == 0) continue;
        uint64_t oTS = announceTS[itid].load();
        if (oTS < lowestTS) {
            lowestTS = oTS;
            tl_otid = itid;
        }
    }
    return lowestTS;
}

// returns timestamp of thread holding the write-lock (if taken)
uint64_t getTSOfWLock(uint32_t widx) {
    uint64_t lowestTS = NO_TIMESTAMP;
    uint16_t wstate = wlocks[widx].load();
    if (wstate != UNLOCKED && wstate != tid) {
        uint16_t otid = wstate;
        uint64_t oTS = announceTS[otid].load();
        if (oTS < lowestTS) {
            lowestTS = oTS;
            tl_otid = otid;
        }
    }
    return lowestTS;
}

uint64_t ribit(uint32_t widx) { return (1ULL << ((widx) % 64)); }
```

as gdb to debug issues in the user's transaction code with 2PL and 2PLSF.

2.8 Irrevocability

Multi-writer concurrency controls are subject to conflict-restarts, causing each transaction to possibly restart multiple times, a term sometimes named *speculative* execution. To address this issue, some concurrency controls provide *irrevocability*, the property that the transaction will *never restart* during its execution. By definition, multiple irrevocable read-only transactions can be executed concurrently, however, a single irrevocable write transaction can be executed at any given time.

Welc *et al.* [27] have proposed a technique to provide irrevocable transactions for concurrency controls with optimistic reads (such as TL2 and LSA), by having a fallback to acquiring read-locks during the load interposing of the irrevocable transaction, this way bypassing the need to have a read-set validation stage at commit time. However, similarly to other 2PL based approaches with a read-writer lock, these techniques scale poorly due to contention on the locks.

If we're willing to sacrifice starvation-freedom, 2PLSF can provide irrevocable read-only transactions. It suffices that an irrevocable read-only transaction starts by announcing its timestamp as being zero, thus ensuring the transaction will never restart.

2PLSF can be further modified to provide irrevocable write transactions by adding a *zero mutex* which, when acquired, gives the transaction the right to chose a timestamp of zero. This transaction will never restart and at the end of its execution it will release the zero mutex. This approach effectively serializes all irrevocable write transactions as they will be waiting to acquire the zero timestamp.

2.9 Memory Reclamation

When deploying a concurrency control and using the system allocator, extra work must be done to dynamically allocate and de-allocate objects. At commit time, concurrency controls with optimistic accesses must use a safe memory reclamation scheme (SMR), passing each object in the de-allocated list to the SMR. Due to its simplicity, the scheme of choice is typically an epoch-based reclamation (EBR) [6, 12, 30].

For two-phase locking algorithms, such as 2PL and 2PLSF, the objects in the de-allocation log can be *immediately de-allocated* at commit time, without any need to utilize a memory reclamation scheme. This is safe to do because the read-locks taken on every read access, guarantee that whatever pointer was read to de-reference the object, this pointer has

not been modified by another transaction (as doing so would require a write-lock on the pointer) therefore guaranteeing the object is *reachable*.

The fact that 2PL foregoes the need to implement and execute a memory reclamation scheme, reduces the engineering effort when choosing 2PLSF versus approaches with optimistic reads. Moreover, the usage of a memory reclamation scheme can introduce additional memory requirements to store the lists of retired objects and it may cause a performance impact on workloads where transactions execute a large number of allocations and de-allocations. In essence, no matter how efficient in space and time is the memory reclamation scheme deployed, the fact that 2PL and 2PLSF do not need a reclamation scheme is an important advantage over optimistic concurrency controls.

A stronger property than safe memory reclamation is *privatization*. Similarly to other 2PL algorithms [8], 2PLSF provides implicit privatization [20] including implicit proxy privatization [16]. This guarantee stems directly from the pessimistic nature of the transactions and comes without any additional performance cost [8].

3 Evaluation

We now present an evaluation of 2PLSF and compare it with other state-of-the-art STM implementations when applied to transactional data structures, using synthetic benchmarks. We executed these microbenchmarks on a dual-socket 2.50 GHz Intel Xeon E5-2683 v4 with a total of 32 hyper-threaded cores (64 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 will study the throughput of different transactional set data structures when deployed using an assortment of STM implementations. On the leftmost plots, the workload is made of 50% random insertions and 50% removals. The central plots have 10% insertions, 10% removals and 80% lookups. The rightmost plots execute lookups exclusively. Each data point represents the mean over 5 runs of 20 seconds.

3.1 Different RW-Locks

We start with an incremental comparison of different 2PL algorithms and locks.

The data points labeled 2PL-RW in Figure 2 represent a 2PL algorithm where each reader-writer lock is implemented with a single 64-bit word, with 8 bits reserved for the identifier of the thread holding the write-lock and the remaining 56 bits the read-indicator, with one bit reserved per thread holding the read-lock. This implementation supports at most 56 concurrent (reader) threads. Having each thread assigned a specific bit in the lock's variable allows for a quick identification of whether or not the current thread already holds the read-lock, which enables efficient detection of read-after-read scenarios during a transaction. Because of this, having multiple threads read the root pointer of a binary search tree implies contention on the reader-writer lock that protects it, due to the high number of `fetch_add()` instructions executed in the same variable of the lock. This is noticeable in Figure 2 where 2PL-RW (the dark blue line with inverted triangles) is never capable of scaling, even on read-only workloads (rightmost plot). The idea of deploying reader-writer locks in a two-phased locking concurrency control is not new and has been explored previously by Dice *et al.* [8] in the TLRW concurrency control however, as noticeable in Figure 2 and as will be shown later in this section, unless a scalable reader-writer lock is used, 2PL will scale poorly.

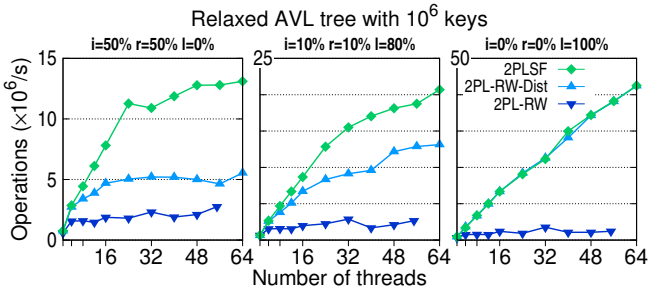

Figure 2. RAVL tree with three different 2PL algorithms.

If an implementation uses instead a reader-writer lock where each thread's read-indicator is on a separate cache line, then it is able to achieve high scalability for read-mostly workloads (light blue line with up triangles, shown as 2PL-RW-Dist in Figure 2). The idea of using separate cache lines for the read-indicators is not new and has been explored previously by Dice *et al.* [2]. To the best of our knowledge, our approach is the first to combine multiple read-indicators per thread, so as to reduce false sharing without increasing the memory footprint.

Our 2PLSF concurrency control described in Algorithm 1 uses the same read-indicator layout as 2PL-RW-Dist, however, it does so in a way as to guarantee starvation-freedom and efficient conflict detection. Similarly to most other concurrency controls, 2PL-RW-Dist uses a backoff strategy to solve conflicts, a strategy referred in the literature as 2PL No-Wait [28, 29], while our 2PLSF does no backoff. As can be observed in Figure 2, on read-mostly workloads (rightmost plot) the 2PLSF approach has no advantage over 2PL-RW-Dist, however, on write-intensive workloads (leftmost plot) 2PLSF is significantly faster at solving conflicts, giving it an advantage of more than 2x over 2PL-RW-Dist. We observed this advantage in all the workloads we executed during our studies and therefore, for the remaining of this paper we will no longer show the results for the 2PL-RW-Dist nor 2PL-RW implementations, although source code for these can be obtained at the github repository of this paper at

http://github.com/pramalhe/2PLSF or through the zenodo link https://zenodo.org/record/7358723#.Y7LQthXMJ4E.

3.2 Set Data Structures

We compared our 2PLSF implementation with the following STMs:

- TL2: the original TL2 implementation [7] with pessimistic writes based on two-phase locking and optimistic read accesses (write transactions require a read-set validation at commit time);
- TinySTM: an STM implementing the LSA concurrency control [11, 12], also with pessimistic writes, read-set validation and optimistic read accesses;
- TLRW-Z: the implementation by Zardoshti *et al.* [30] of the TLRW STM by Dice *et al.* [8], which itself is a variation of two-phase locking (*no-wait*) with reader-writer locks;
- OREC-Z: the Orec STM implementation by Zardoshti *et al.* [30];
- OFWF: the OneFile STM by Ramalhete *et al.* [22] which provides wait-free transactions where write transactions are serialized and do not need a read-set validation, while read accesses are done optimistically.

Although not all papers claim to do so, to the best of our knowledge all these STMs provide *opaque* [14] transactions.

We also compared with Orec-eager and the results were similar to Orec-lazy, therefore we chose to show the later only. We compared against the TL2 implementation provided by Zardoshti *et al.* which, despite being more stable than the original TL2 implementation, was slower, therefore we chose to display results for the fastest of the two.

Not all STMs are shown in all plots, as some implementations were unstable to the point of preventing the collection of enough data for a comparison. Our 2PLSF implementation has been put through a varied set of stress tests, never having detected any loss of invariant nor memory leak.

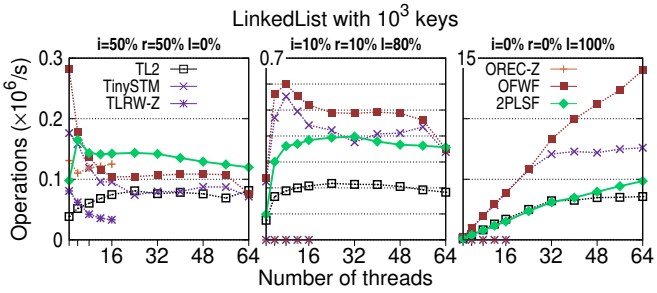

Figure 3. Linked list set with 10^3 keys.

Figure 3 shows the throughput for a transactionally annotated linked list set under the different STMs. In write-intensive workloads (leftmost plot) 2PLSF is usually the winner however, on other workloads it is surpassed by OneFile and TinySTM as these two STMs have optimistic loads.

The cost of incrementing a global clock on every write transaction is the main reason behind the lack of scalability of TL2 and TinySTM in the hash set shown in Figure 4. In those concurrency controls, read-only transactions do not increment the global clock and therefore, read-only workloads can scale effortlessly. When the write transaction is longer, like the case of inserting/removing a node in a skip list (Figure 5) or a zip tree (Figure 6), the increment of the global clock is no longer a bottleneck, because the absolute number of transactions is low enough that executing one fetch-and-add on the global clock per transaction does not create significant contention. In 2PLSF, only conflicting transactions need to increment a global clock, giving 2PLSF high scalability even in short disjoint write transactions, like write intensive workloads of the hash set or the relaxed AVL tree (Figure 7).

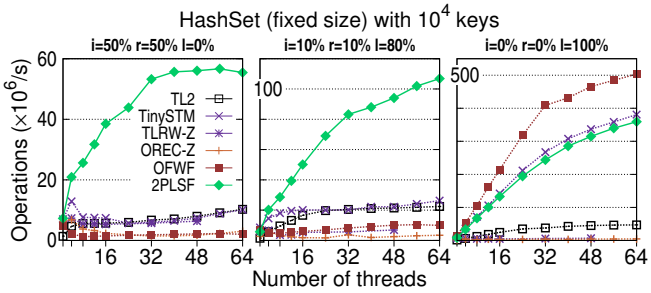

Figure 4. Hash set with 10^4 keys.

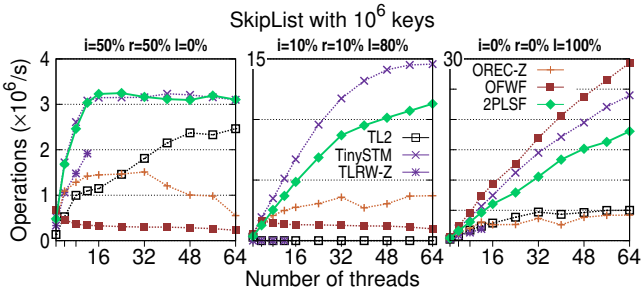

Figure 5. Skip list with 10^6 keys.

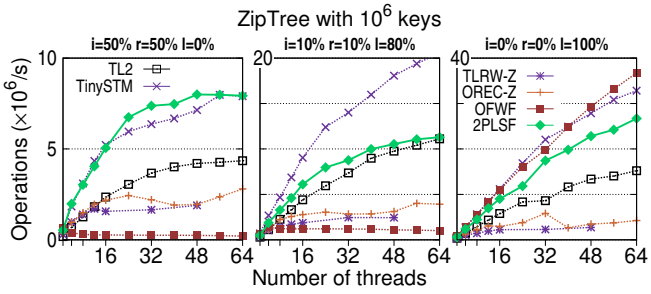

Figure 6. Zip tree with 10^6 keys.

Figures 5 and 6 show a skiplist and ziptree with 10^6 keys. For both data structures TinySTM and 2PLSF provide the highest throughput in write intensive workloads, with

TinySTM having a clear advantage on the read-mostly workloads due to its optimistic read accesses.

Figure 7 shows another transactional data structure, the relaxed AVL tree [15], a data structure specifically designed for disjoint access. This balanced tree provides absolute higher throughput than the other trees. On the write-intensive workload 2PLSF is the stabler of the STMs as it is capable of quickly ordering transactions under conflict. For the other two plots (center and rightmost) the advantage of the optimistic accesses of TinySTM once again comes into play, surpassing 2PLSF.

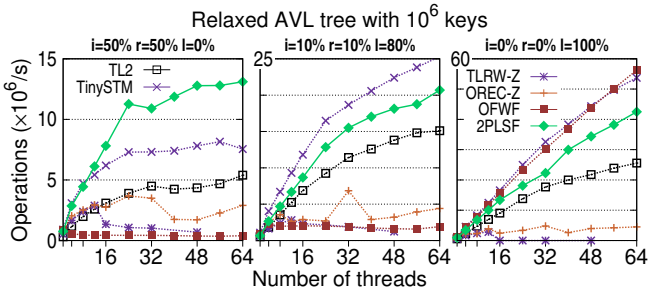

Figure 7. RAVL tree with 10^6 keys.

3.3 Map Data Structures

In the following microbenchmark, instead of using *set* implementations we use key/value *map* implementation where the *value* is a pointer to a record containing 100 bytes of user-generated data. We randomly chose to either insert a key/value mapping ($i = 1\%$), remove a key/value mapping ($r = 1\%$) or execute an update on a record by doing a random key lookup and modify the contents of the record ($u = 98\%$). This workload consists solely of write transactions but because the number of insertions and removals of nodes in the data structure are small, the operations are more disjoint than on the corresponding set microbenchmark with $i = 50\%, r = 50\%$. The results can be observed in Figure 8. If we compare the three plots in Figure 8 with the leftmost

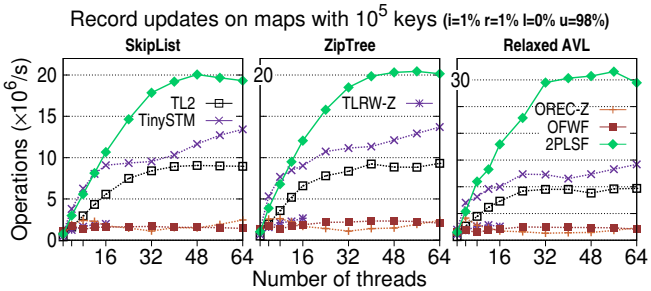

Figure 8. Three maps with 10^5 keys.

plots of Figures 5, 6 and 7, we observe that for the high update workload 2PLSF has a significant advantage over the others. The reason is mostly related to TinySTM and TL2 always incrementing a global clock in write transactions, which means that the scalability for a high-update workload reaches a plateau imposed by the number of increments that the CPU can execute on an atomic variable (the global clock) per second. Because of this factor, in the three plots of Figure 8 the throughput of those STMs never goes above 15 million transactions per second.

2PLSF is not encumbered by this effect because it has no global clock. It does increment a shared atomic variable to decide the order of the transactions however, it does so only in the event of a conflict. In Figures 5, 6 and 7 the write-intensive workloads are not disjoint enough to allow a higher scalability, particularly on the skip list where the throughput stays below 4 million transactions per second, but on the more disjoint workloads (like Figure 8) the advantage of 2PLSF becomes significant. The increment of a global clock for conflicts may seem a minor detail however, assigning a unique number to each transaction implies an increment on a centralized atomic variable which can be a significant scalability bottleneck, even more so if it were to do that for read-only transactions like it is done in 2PL *wait-or-die*. One of TL2 and TinySTM's main strengths is the capability of executing read-only transactions *without* incrementing the global clock. 2PLSF goes beyond this by incrementing it solely for write-transactions with conflicts.

3.4 Tail Latency

The data structures shown in the previous section are important as they are representative of the indexing used in databases however, concurrency controls are used in many other scenarios and some of these are particularly prone to conflicts. In the following synthetic benchmark we allocate an array of counters where threads execute pair-wise conflicting transactions. The first two threads will increment the first 20 counters, with the first thread starting from index 0 of the array until index 19, while the second thread starts from index 19 to 0. As illustrated in Figure 9, the next two threads will write into the next 20 counters, with the third thread starting from index 20 until 39, while the fourth thread starts from 39 to 20, and so on. This workload is particularly prone to conflicts, however, it should theoretically allow for scalability as one thread of each pair *should* be able to commit its transaction.

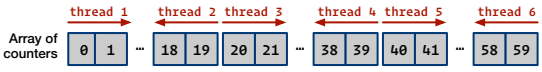

Figure 9. Principle of the latency benchmark.

We executed this benchmark for 20 seconds, collecting the duration of each transaction so as to compute different percentiles of the latency distribution. In Figure 10 we show the throughput for this workload (leftmost plot, higher is better) and we show the percentile latency P90 (center plot, lower is better) and P99 (rightmost plot, lower is better).

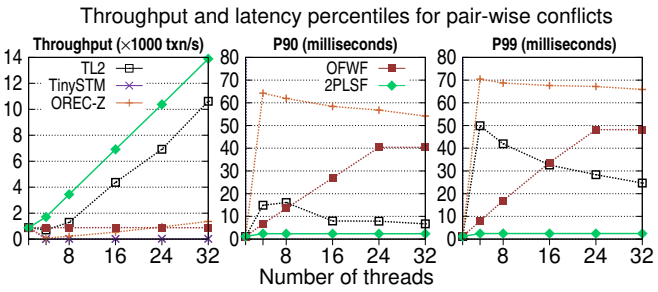

Figure 10. Throughput (left) and Latency (center and right).

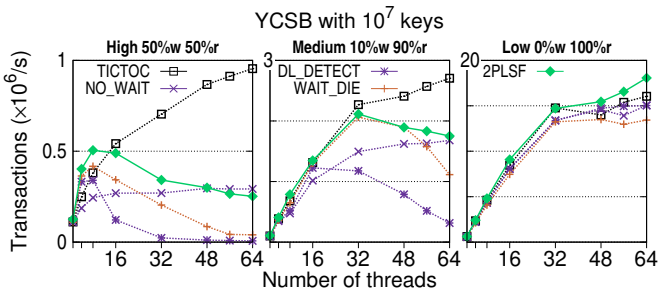

Figure 11. Three different workloads of YCSB with 10^7 keys.

As we can see from the figure, 2PLSF and TL2 are able to scale in this scenario (leftmost plot) with 2PLSF maintaining a strong cutoff on latency, with the P99 being 2.4 milliseconds and the maximum latency (slowest transaction taking) 4.4 milliseconds. On the other hand, TL2 is also capable of scaling for this workload, but its P99 reaches 50 milliseconds with the lengthiest transaction taking 5.6 seconds to complete. For TinySTM the P99 is 5 seconds and therefore not visible in the latency plots, with its lengthiest transaction taking 16.8 seconds to complete. Even for OneFile, which guarantees wait-free progress, the P99 grows almost linearly with the number of threads, with its lengthiest transaction taking 92 milliseconds to complete, an order of magnitude worse than 2PLSF. OneFile aggregates all in-flight transactions into a single execution, causing latency to grow proportionally to the number of competing threads. This benchmark indicates a clear advantage for 2PLSF in workloads with pair-wise conflicts.

3.5 YCSB in DBx1000

We have integrated our 2PLSF concurrency control into the DBx1000 benchmark [28] in order to evaluate it with other concurrency controls commonly used in DBMS. In this particular implementation of YCSB, the index is not protected by the transaction, being a purely sequential hashmap data structure. This is possible because this benchmark executes only record updates, inserting all records initially in the table during a prefilling phase, never removing nor inserting records during the measurement phase. This means that transactions in this benchmark do not need to be opaque and in fact, recent work has shown that the highest scalable concurrency control for this benchmark is TicToc [29], a concurrency control which is serializable but not opaque.

We've executed the YCSB [5] benchmarks as described by Yu *et al.* [29] and show the results in Figure 11. We have disabled the abort buffer (which places conflicting transactions in a temporary buffer to try them at a later time) and the restart backoff, as those would be incompatible with the 2PLSF algorithm.

From left to right, the plots show workloads with high, medium and low contention. In the high contention workload, 2PLSF is able to match and even surpass TicToc initially but loses scalability at 16 threads and beyond. We believe this happens because TicToc allows for more interleaving executions to commit and therefore, transactions have less conflicts, thus showing the performance advantage of serializability over opacity in high contention scenarios. Due to the reduced number of conflicts on the medium contention workload, 2PLSF is capable of providing throughput similar to TicToc up to 32 threads, being better than the other 2PL based algorithms for higher thread counts. On the low contention workload, 2PLSF provides high scalability on all thread counts, with a slight advantage over TicToc, which after profiling can be explained by the cost performing the copy of the data. Optimistic concurrency controls like TicToc must copy the 100 byte-sized tuple to a temporary memory location and then post-validate the timestamp on the lock protecting the tuple, while 2PL based algorithms can directly read from the tuple without performing the extra step of copying it. The performance advantage of TicToc in conflicting workloads comes with a price: Transactions in TicToc are serializable, being validated at commit time and therefore, if we apply TicToc to a transactional data structure, the invariants of the data structure may no longer hold, resulting in incorrect behavior, such as crashes or infinite loops [23].

Figure 11 also shows the three variants of two-phase locking which are part of the DBx1000 benchmark suite, namely *no-wait*, *wait-or-die* and *deadlock-detection*, all implemented with `pthread_mutex_t` as the underlying lock.

4 Related Work

Starvation-free STMs have been shown before in the literature [3, 13, 24, 26].

KSFTM [3] is one example of a starvation-free concurrency control. Unfortunately this implementation is object-based which makes a comparison difficult. Moreover this implementation takes a global lock to execute mutative transactions and this lock is implemented with a `std::mutex` which is neither scalable nor starvation-free.

TM2C [13] is another example. When using the FairCM contention manager, TM2C provides starvation-freedom by assigning a priority to each transaction which does not change during its lifespan. The priorities define a total order on the set of concurrent transactions

A detailed study on multiple contention managers, some of which are starvation-free, was shown by Spear *et al.* [24].

PLOR [4] is a recently presented starvation-free concurrency control. All transactions in PLOR must take a unique timestamp which, as we have shown, is a significant performance bottleneck. Moreover, in PLOR the read-lock acquisition creates a list of readers based on arrival time, thus implicitly serializing readers contenting on the same lock, which will hamper scalability for workloads with non-disjoint reads. Such an effect is not observable on workloads like YCSB because PLOR uses MassTree [18] as the indexing data structure (which itself uses an optimistic concurrency control for reads), but it will certainly become observable if PLOR is applied to a tree data structure, like the ones shown in Figures 5,6,7,8.

4.1 2PL Wait-Or-Die

The main idea of 2PLSF is to order the transaction as soon as a lock conflict arises and use this order to resolve the conflict. This is reminiscent of the classical 2PL *Wait-Or-Die* strategy first described by Bernstein et al. [1] however, it differs from it on one vital aspect. The *Wait-Or-Die* strategy imposes that transactions have a unique sequence number (the transaction's *priority* [1]) to order *all* the transactions, while in 2PLSF we use a unique sequence number to order only the conflicting transactions. This may seem a minor detail however, as we have seen in section 3, assigning a unique number to each transaction implies an increment on a centralized atomic variable which can be a significant scalability bottleneck, even more so if we were to do that for read-only transactions like it is done in *Wait-Or-Die*.

4.2 2PL No-Wait

In the No-Wait variant of 2PL, when a conflict occurs, the transaction aborts and will restart after a *backoff* period of time. This backoff period may be a fixed interval, or it may be derived from an heuristic, like an exponential backoff formula. Given that each transaction may be retried without any guarantee of success, the No-Wait algorithm is not starvation-free, and in fact, it's not even live-lock free, as two threads may repeatedly acquire locks in opposite order, preventing each other from committing.

5 Discussion

For read-mostly workloads, it is unlikely that pessimistic concurrency controls like 2PL/2PLSF will ever be able to beat concurrency control algorithms with optimistic read accesses, like TL2 or LSA. The lower synchronization cost of ordering load instructions (STMs with optimistic reads), relative to the synchronization cost of store-load ordering (pessimistic STMs), is likely to remain true on modern CPUs for the foreseeable future, implying that the advantage of the optimistic approach will continue to hold for read-dominated workloads. However, it is hard to imagine an algorithm that simultaneously provides optimistic reads *and* starvation-freedom, or even live-lock freedom, without at least a pessimistic fallback mechanism. Providing starvation-freedom is vital to guarantee low tail latency of each individual transaction under a multitude of workloads.

With 2PLSF we have shown that when using efficient reader-writer locks, pessimistic concurrency controls can provide high read throughput, lagging not far behind the state of the art in optimistic concurrency controls, while maintaining starvation-freedom. Unlike TL2 and LSA which require a global clock to be incremented on every write transaction, write transactions in 2PLSF increment a global clock solely when conflicts arise, allowing disjoint write transactions to scale unimpeded.

Today, the majority of DBMS utilize 2PL to manage concurrent access to the database's records, however we know of no commercial DBMS which uses 2PL for its indexing data structures. This separation of concurrency controls means that operations on the indexing data structure are not part of the transaction. The rational for this design decision is clear, as adding the indexing data structure to the transaction with any of the current 2PL approaches would significantly hamper scalability, which in turn has created the myth that 2PL by itself has poor scalability.

Thanks to its novel algorithm and reader-writer lock, 2PLSF is scalable enough that even the indexing data structure can be made part of the transaction without sacrificing scalability nor consistency. Moreover, its efficient conflict resolution and starvation-freedom progress, imply its transactions will have reduced tail latency in a multitude of workloads, making this an ideal concurrency control for database implementers.

Acknowledgments

This work is supported in part by the Swiss National Science Foundation (SNSF) under project number 200021-178822/1.

References

[1] 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

[2] 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

[3] 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

[4] 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

[5] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[6] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. 2010. NOrec: streamlining STM by abolishing ownership records. 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, 67–78. https://doi.org/10.1145/1693453.1693464

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

[15] Kim S. Larsen. 1994. AVL Trees with Relaxed Balance. In *Proceedings of the 8th International Symposium on Parallel Processing, Cancún, Mexico, April 1994*, Howard Jay Siegel (Ed.). IEEE Computer Society, 888–893. https://doi.org/10.1109/IPPS.1994.288201

[16] Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. 2009. Anatomy of a scalable software transactional memory. In *Proc. 4th ACM SIGPLAN Workshop on Transactional Computing*.

[17] 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

[18] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. 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, 183–196. https://doi.org/10.1145/2168836.2168855

[19] 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

[20] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. 2008. Single global lock semantics in a weakly atomic STM. *ACM SIGPLAN Notices* 43, 5 (2008), 15–26. https://doi.org/10.1145/1402227.1402235

[21] Jonathan M. Nash. 1999. A scalable and starvation-free concurrent locking mechanism. *Concurr. Pract. Exp.* 11, 13 (1999), 823–833.

[22] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. 2019. OneFile: A Wait-Free Persistent Transactional Memory. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*. IEEE, 151–163. https://doi.org/10.1109/DSN.2019.00028

[23] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. 2019. Fast General Distributed Transactions with Opacity. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 433–448. https://doi.org/10.1145/3299869.3300069

[24] 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

[25] S. Swaminathan, J. Stultz, J. F. Vogel, and Paul E. McKenney. 2002. Fairlocks A High Performance Fair Locking Scheme. In *International Conference on Parallel and Distributed Computing Systems, PDCS 2002, November 4-6, 2002, Cambridge, USA*, Selim G. Akl and Teofilo F. Gonzalez (Eds.). IASTED/ACTA Press, 241–246.

[26] 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

[27] 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

[28] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.*

8, 3 (2014), 209–220. https://doi.org/10.14778/2735508.2735511

[29] 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

[30] 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