TLRW: Return of the Read-Write Lock

Dave Dice

Sun Microsystems Laboratories, 1 Network Drive,
Burlington, MA 01803-0903 USA
dave.dice@sun.com

Nir Shavit

Tel-Aviv University, Tel-Aviv 69978, Israel and Sun
Microsystems Laboratories, 1 Network Drive,
Burlington, MA 01803-0903 USA
shanir@cs.tau.ac.il

Abstract

TL2 and similar STM algorithms deliver high scalability based on write-locking and invisible readers. In fact, no modern STM design locks to read along its common execution path because doing so would require a memory synchronization operation that would greatly hamper performance.

In this paper we introduce TLRW, a new STM algorithm intended for the single-chip multicore systems that are quickly taking over a large fraction of the computing landscape. We make the claim that the cost of coherence in such single chip systems is down to a level that allows one to design a scalable STM based on read-write locks. TLRW is based on byte-locks, a novel read-write lock design with a low read-lock acquisition overhead and the ability to take advantage of the locality of reference within transactions. As we show, TLRW has a painfully simple design, one that naturally provides coherent state without validation, implicit privatization, and irrevocable transactions. Providing similar properties in STMs based on invisible-readers (such as TL2) has typically resulted in a major loss of performance.

In a series of benchmarks we show that when running on a 64-way single-chip multicore machine, TLRW delivers surprisingly good performance (competitive with and sometimes outperforming TL2). However, on a 128-way 2-chip system that has higher coherence costs across the interconnect, performance deteriorates rapidly. We believe our work raises the question of whether on single-chip multicore machines, read-write lock-based STMs are the way to go.

1. Introduction

STM design has come a long way since the first STM algorithm by Shavit and Touitou (Shavit and Touitou 1997), which provided a non-blocking implementation of static transactions (see (Shavit and Touitou 1997; Ennals; Harris and Fraser; Lev et al. 2008; Marathe et al. 2006; Moir 2004; Saha et al. 2006; Dice et al. 2006; Riegel et al. 2006a; Menon et al. 2008; Spear et al. 2008)). A fundamental paper by Ennals (Ennals) suggested that on modern operating systems, deadlock avoidance is the only compelling reason for making transactions non-blocking, and that there is no reason to provide it for transactions at the user level. The use of locks eliminates the need for indirection and shared transaction records as in non-blocking STMs (Harris and Fraser; Moir 2004). Deadlocks and livelocks are dealt with using timeouts and the ability of transactions to request other transactions to abort. Ennals's view was quickly seconded by Dice and Shavit (Dice and Shavit 2007) and by Saha et al (Saha et al. 2006). The final barrier to the acceptance of such lock-based algorithms was removed with the introduction of global clock based consistency by Dice, Shalev, and Shavit (Dice et al. 2006) (the idea of using a global clock for internal consistency was independently proposed by Reigel, Felber, and Fetzer in the context of their non-blocking Snapshot Isolation STM and LSA algorithms(Riegel et al. 2006a,b)).

Today, most, if not all new lock-based STMs use a variation of the TL2/LSA style global-clock algorithm using invisible reads. When we say invisible reads, we mean that the STM does not know how many readers might be accessing a given memory location. The drawback of invisible read based STMs are the overheads associated with maintaining and validating a read-set of locations (Dice et al. 2006), and the unacceptably high cost of providing *implicit privatization* (Menon et al. 2008) and *proxy privatization*[1] (Lev et al. 2008; Marathe 2008). One should note that STMs that use centralized data structures, such as RingSTM (Spear et al. 2008) or compiler assisted coarse grained locking schemes (Menon et al. 2008), can provide implicit privatization without the need for explicit visible readers.

Despite these drawbacks, using invisible reads is compelling since visible reads require that the number of readers, or at the very least, the existence of readers, be recorded per memory location. This is a task that on the face of it requires a relatively expensive synchronization per read operation. There are novel mechanisms such as the scalable non-zero indicators (SNZI) of Ellen et al (Ellen et al. 2007), that greatly reduce the synchronization overhead of detecting readers. Unfortunately, SNZI at the very least requires a CAS per increment or decrement operation. Moreover, when contended it requires a distributed tree of cache-line independent nodes leading to the "indicator" location. This is an unacceptable space complexity in practice.

1.1 Our New Approach

In this paper, we examine STM design in the context of multicore systems-on-a-chip, a class of architectures that is already common in the server space and is rapidly taking over the desktop computing space. For such systems, we claim that the cost of coherence is down to a level that suggests another way to approach the problem:

[1] Proxy privatization is the case of implicit privatization where one thread's transaction privatizes an object that is then used privately by another thread. This might, in some cases, turn out to be harder than privatizing for a thread's own use.

designing visible-reader based STMs using *read-write* locks.[2]. We call our new read-write lock based STM design *TLRW*, and its key algorithmic technique, the *bytelock*.

Why design an STM based on read-write locking? Because the overall design is significantly simpler and more streamlined than invisible read based STMs like TL2, TinySTM (Felber et al. 2008), or McRT (Saha et al. 2006), in which one locks only written locations and validates coherence of the ones being read. In contrast, in a read-write lock-based STM, a transaction locks every location before either reading or writing. Then, upon completion, it releases all the locks. As we argue, this simple design confers amazing benefits:

- No costly validation of the read and write set (what you lock is what you get).
- Stronger progress properties (especially for long transactions) than invisible read based STMs such as TL2.
- Implicit privatization including implicit proxy privatization.
- Support for irrevocable transactions (Welc et al. 2008).

Our TLRW design provides the latter two properties naturally and with virtually no overhead.

However, a problem remains. Even on multicore chips, existing read-write lock designs that count readers simply do not scale because of the synchronization overheads. To overcome this problem, we introduce *bytelocks*, a new class of high performance read-write locks designed to deliver scalable performance in the face of high levels of read-lock acquisition.

The idea behind bytelocks is in itself very simple, and we ourselves were surprised at the scalable performance they deliver in the context of TLRW transactions. In a nutshell, in a bytelock, we split the lock record into an array of bytes, one per thread. On modern AMD, Intel, and Sun processors, these bytes can be written individually and read in batches. Each thread is assigned a byte, which it uses as a flag indicating it is reading the location. The byte is set using a simple store followed by a store-load memory barrier. This has the advantage of avoiding CAS operations that typically have excessive local latency, can be interfered with and require a retry, and on systems such as Niagara, may incur a cache invalidation (Dice 2008). As we show, the benefit of this design is scalable performance in the common case. The lock-word also contains a 32 bit counter that is incremented or decremented using CAS by all reader threads that were not assigned a byte in the byte-array. On current architectures one can use 48 bytes that align on a single cache line (or 112 that align on two cache lines with little performance loss).

We support our thesis, that read-write lock-based STMs are a viable approach on state-of-the-art single chip multicore systems, through a series of benchmarks. These are, unfortunately, standard micro benchmarks and not real applications, but we hope they will suffice to convince the reader of the benefits and drawbacks of our design. We tested TLRW on a single chip UltraSPARC® T2 (Niagara II) multicore machine. Our results indicate that TLRW, which always provides implicit privatization, often matches and sometimes outperforms TL2, and always outperforms a version of TL2 with implicit privatization. In some cases, such as for long transactions, TLRW has stronger progress properties than TL2 because the only source of aborts are time-outs, so chances are better that a transaction does not abort after much of it has already completed.

We also tested TLRW on a 128-way Enterprise T5140® server (Maramba) machine, a 2-chip Niagara system, which has relatively high inter-chip coherence costs. Here, as expected, the performance of TLRW was consistently inferior to that of TL2, though in some cases it matched that of TL2 with privatization. Our conclusion, as we hope the reader will agree, is that TLRW suggests a new design direction for lock-based STMs. We believe this direction will become increasingly viable as the cost of coherence on multicore systems drops.

The next section describes our new TLRW algorithm in detail. We then provide a performance section that analyzes its behavior.

[2] The trend with new multicore processors by all main manufacturers seems to be towards lower synchronization and coherence costs.

2. Read-Write Lock Based Transactional Locking

The TLRW algorithm we describe here is a simple one phase commit style algorithm using read-write locks. This means that threads acquire locks for reading as well as for writing. This approach, by its very nature, guarantees internal consistency (Dice et al. 2006) and implicit privatization (Menon et al. 2008). As we will show, it also allows for a simple implementation of irrevocable transactions (Welc et al. 2008). Finally, it avoids some of the performance overheads of invisible-read based STMs such as TL2 (Dice et al. 2006) and TinySTM (Felber et al. 2008), since read sets do not record values and there is no need for read-set validation.

Unfortunately, STMs using naive read-write locks have abominable scalability since reading a location requires an update of a "read counter," which requires a CAS operation. Thus, locations that are shared by multiple readers (such as the root of a red-black tree) become hotspots and cause a deterioration in performance.

The claim we wish to make in this paper is that on new multicore machines, as long as one remains on chip (i.e. low coherence costs), using read-write locks is a viable approach if one can get low overhead read-write locks.

2.1 Read-write Byte-locks

The key idea in our new TLRW algorithm is the use of a new class of read-write lock which we call a *byte-lock*. The bytelock is directed at minimizing read-lock acquisition overheads. The basic lock structure consists of 64 bytes aligned across a single cache line and logically split into three distinct zones: an *owner* field, a *byte-array*, and a *read-counter*. The *owner* field is set to the thread id of the writer owning the lock and is set to 0 if no writer holds the lock. In the most basic implementation the *byte array* consists of $k = 48$ bytes, one per reader thread with the lower k ids. We will call these k threads the *slotted* threads and the remaining $n - k$ (where n is the total number of threads in the system) the *unslotted* threads. The algorithm will be highly effective for slotted threads and have standard read-write lock performance for the unslotted ones. The third field is a 32-bit *reader-count* of the number of current reader threads used by unslotted reader threads.

Here is how the *byte-lock* is used to implement a read-write lock by a thread i.

- *To acquire a lock for writing*: thread i uses a CAS to set the owner field from 0 to i. If the field is non-0 there is another owner, so i spins, re-reading the owner field until it is 0. Once the owner field is set to i, it spins until all readers have drained out. To do so, if i is slotted, it sets its reader byte to 0 (just in case it was already a reader). If i is unslotted, it checks a local indicator (such as a transaction's read-set) to determine if it is a reader and decrements its reader-count if it is. In both cases, slotted and unslotted, i then spins until all of the locations of the byte array and the reader-count are 0. Spinning is efficient since one can read 8 bytes of the lock word at a time (on SPARC, more on Intel). To release the write-lock simply store a 0 into the owner field.
- *To acquire a lock for reading*: We implement the lock following the flag principle (Herlihy and Shavit 2008). Readers store their

own byte and then fetch and check the owner, while writers CAS the owner field (we CAS to resolve writer vs. writer conflicts) and then fetch all the reader bytes. In detail:

- If the thread i is slotted then: if i is the owner or the ith byte in the byte array is set, proceed. Otherwise, store a non-zero value into the ith byte and execute a memory write barrier (no use of CAS). Sparc, Intel, and AMD architectures allow byte-wise stores. If the owner field is non-0, store 0 into the ith byte and spin until the owner becomes 0. In other words, writers get precedence. Repeat until the ith byte is set and no owner is detected. To release, store a 0 to the ith byte field. There is no need for a memory barrier instruction.
- If thread i is unslotted then: if i is the owner or a local indicator (such as a transaction's read-set) indicates it is a reader, then proceed. Otherwise, increment the reader-count by 1 using a CAS. Check the owner, and if it is non-0 use a CAS to decrement the read-counter by 1. Repeat until after the reader-count is incremented, no owner is detected. To release, decrement the reader count field using a CAS.

Finally, we note that we allow read-write locks to time-out while attempting to acquire the lock. If lock acquisition times out the thread aborts the transaction and returns an appropriate indication.

The size of the byte-array is based on 64 byte AMD, Sun, or Intel architectures. One can extend k to 112 threads by allowing the lock to extend into a second cache line at the cost of an additional cache access upon read (to be explained later).

The important feature of the new byte-lock is that unlike standard read-write locks, for all slotted threads, reading a location protected by a byte-lock requires a store followed by a memory barrier instruction. It thus avoids a *CAS on the same location* for any of the k slotted threads. CAS has typically high local latency. More importantly perhaps, it is optimistic and can be interfered with and require a retry (one thread's success is bound to cause the next thread to fail), and on systems such as Niagara, may incur a cache invalidation (Dice 2008). For unslotted readers, the byte-lock behaves like a normal read-write lock, with threads CASing the same read-counter.

Notice that for slotted threads, there are additional performance benefits. There is no need for a writer to separately track if it is a reader, which means that when used in an STM, it will not have to traverse the read set except to release locks at the end of a transaction. There is also no need for second memory barrier instruction to set the read byte to 0, a thread can simply wait for the processor's pipeline flush. This saves a CAS in many cases relative to standard read/write locks.

2.2 The Basic TLRW byte-lock Algorithm

In our TLRW design, we associate a byte-lock with every transacted memory location (one could alternately use a byte-lock per object). We stripe the locks across the memory, so that multiple locations share the same lock. This saves space but can lead to false write conflicts in a manner similar to (Dice et al. 2006; Zilles and Rajwar 2007). We maintain thread local read- and write-sets as linked lists. These sets track locations on which locks are held. The write set contains undo values since our algorithm will store new values in-place, but it should be noted that our algorithm could support a redo log as well, in which case read-locks would be acquired during the speculative execution phase and write-lock acquisition would be deferred until commit-time.

We now describe the basic TLRW algorithm. Unlike TL2, TLRW does not require safe loads. The following sequence of operations is performed by a *transaction*, one that performs both reads and writes to the shared memory.

1. **Run through a an execution:** Execute the transaction code. Locally maintain a *read-set* of addresses loaded and an *undo write set* of address/value pairs stored. This logging functionality is implemented simply by augmenting loads with instructions that record the read address and replacing stores with code recording the address and value to-be-written in case the transaction must abort.[3]

   The transactional read attempts to acquire a location's read-lock. (As an optimization it can delay waiting for a bus lock to be released). If the acquisition is successful, it reads the location, records the location in the read-set and returns the location's value. Similarly, a transactional write acquires the location's write lock, records the current value in the undo set, and writes the value to the location.
2. **Time out abort:** The only source of *aborts* is a time out by some thread while attempting to acquire a lock. In such a case, threads use the undo write log to return all locations to their pre-transaction values. It then releases all the read and write locks it holds.
3. **Commit** release the write locks and then the read locks.

The beauty of this algorithm in comparison to most STM algorithms in the literature, is its simplicity. The only reason to abort transactions is deadlock avoidance, which makes for a very strong progress property. Other more elaborate schemes, such as detecting cycles in a 'waits for' graph are also possible and may be worthwhile in some contexts.

The following safety properties follow almost immediately from the fact that a transaction holds locks on all locations it reads or writes.

LEMMA 2.1. *TLRW Transactions are internally consistent (i.e. operate on consistent states (Dice et al. 2006; Guerraoui and Kapalka 2008)), and are externally consistent (i.e. are serializable (Herlihy and Wing 1990)).*

LEMMA 2.2. *TLRW Transactions provide implicit privatization and implicit proxy privatization.*

In terms of liveness, from the fact that byte-locks are deadlock-free and eventually transactions time out, it follows that

LEMMA 2.3. *TLRW Transactions never deadlock.*

In terms of lockout-freedom, guarantees are similar to those of the TL2 algorithm in the sense that livelocks can happen only if transactions time-out again and again. However, notice that here transactions do not cause each other to repeatedly abort by invalidating each other's read set. Livelocks can happen only if some threads are slow to release locks. To lower the chances of such livelocks, we use an exponential backoff scheme on the completion time, the delay before a transaction is timed-out. Notice that we add spinning to byte-lock acquisition attempts only as an optimization, while exponentially backing off on the completion time is crucial.

2.3 Irrevocable Transactions

A further benefit of TLRW is that one can readily implement irrevocable transactions. Irrevocable transactions, introduced by (Welc et al. 2008), are transactions that never abort, and can be used in case the transaction contains an I/O operation or is long and will never complete in an optimistic fashion (a hash table resize or an iterator call on a search structure). We use the "irrevocable

[3] Notice that there is no need for non-faulting loads or trap handlers. In TL2 one had to use a non-faulting load as a transaction fetch may have loaded from a just privatized region that had been made unreachable.

transaction" approach best outlined in a paper by Welc et al (Welc et al. 2008; Ni et al. 2008), albeit in a much simpler fashion, and with a stronger progress guarantee.

The idea outlined by Welc et al is simple. We will guarantee that there is always no more than one active irrevocable transaction, allowing some active irrevocable transaction to complete. This is done by maintaining a global *irrevocable-bit* or, to guarantee stronger progress, an *irrevocable-lock* consisting of a CLH queue-lock (Craig 1993; Magnussen et al. 1994). Any irrevocable transaction sets the bit (alternately attempts to acquire the CLH lock) using a CAS. Once the bit is set (alternately the CLH lock is acquired), the transaction proceeds without ever timing out. If a deadlock situation arises, the *revocable* transactions involved in it will eventually time out and free the locations that will allow the single irrevocable transaction to proceed. Notice that by using a CLH lock, we can guarantee FCFS order on the irrevocable transactions so they are guaranteed to never starve. While such transactions are in progress, all revocable transactions that do not overlap in memory can proceed as usual.

The overhead of the irrevocable bit mechanism is minimal since transactions are spinning locally, and if one deals with long transactions, the CLH lock can be replaced by a monitor style lock that allow transactions to sleep while they are queued (the overhead of such a lock will be mitigated by the transactions cost, say, the cost of an I/O operation or its being long.

We added to TLRW the ability to automatically shift a transaction into irrevocable mode after some number of back to back failures but have yet to fully benchmark it. Note that for simplicity we have such transactions abort before retrying in irrevocable mode though one could think of other policies that allow conditional transition during the transaction.

3. Empirical Performance Evaluation

This section presents a comparison of our TLRW algorithm using byte-locks to algorithms representing state-of-the-art lock-based (Ennals) STMs on a set of microbenchmarks that include the now standard concurrent red-black tree structure (Herlihy et al. 2003) and a randomized work-distribution benchmark in the style of (Shavit and Touitou 1997).

The red-black tree tested with was derived from the `java.util.TreeMap` implementation found in the Java 6.0 JDK. That implementation was written by Doug Lea and Josh Bloch. In turn, parts of the Java TreeMap were derived from the Cormen et al (Cormen et al. 2001). We would have preferred to use the exact Fraser-Harris red-black tree but that code was written to to their specific transactional interface and could not readily be converted to a simple form.

The red-black tree implementation exposes a key-value pair interface of *put*, *delete*, and *get* operations. The put operation installs a key-value pair. If the key is not present in the data structure put will put a new element describing the key-value pair. If the key is already present in the data structure put will simply insert the value associated with the existing key. The get operation queries the value for a given key, returning an indication if the key was present in the data structure. Finally, delete removes a key from the data structure, returning an indication if the key was found to be present in the data structure. The benchmark harness calls put, get and delete to operate on the underlying data structure. The harness allows for the proportion of put, get and delete operations to be varied by way of command line arguments, as well as the number of threads, trial duration, initial number of key-value pairs to be installed in the data structure, and the key-range. The key range of 2K elements generates a small size tree while the range of 20K elements creates a large tree, implying a larger transaction size for the set operations. We report the aggregate number of successful transactions completed in the measurement interval, which in our case is 10 seconds.

In the random-array benchmark each worker thread loops, generating a random index into the array and then executes a transaction having R reads, W writes, and RW read-modify-write operations. (The order of the read, write, and read-write accesses within a transaction is also randomized). The index is selected with replacement via a uniform random number generator. While overly simplistic we believe our model still captures critical locality of reference properties found in actual programs. We report the aggregate number of successful transactions completed in the measurement interval, which in our case is 10 seconds.

For our experiments we used 64-way Sun UltraSPARC® T2 multicore machine running Solaris™ 10. This is a machine with 8 cores that multiplex 8 hardware threads each and share an on chip L2 cache. We also used a 128-way Enterprise T5140® server (Maramba) machine, a 2-chip Niagara system.

In our benchmarks we "transactified" the data structures by hand: explicitly adding transactional load and store operators, but ultimately we believe that compilers should perform this transformation. We did so since our goal is to explore the mechanisms and performance of the underlying transactional infrastructure and not the language-level expression of "atomic." Our benchmarked algorithms included:

Mutex We used the Solaris POSIX threads library mutex as a coarse-grained locking mechanism.

TL2 The transactional locking algorithm of (Dice et al. 2006) using the GV4 global clock algorithm that attempts to update the shared clock in every transaction, but only once: even if the CAS fails, it continues on to validate and commit. We use the latest version of TL2 which (through several code optimizations, as opposed to algorithmic changes) has about 25% better single threaded latency than the version used in in (Dice et al. 2006). This algorithm is representative of a class of high performance lock-based algorithms such as (Saha et al. 2006; Welc et al. 2008; Felber et al. 2008).

TL2-IP A version of TL2 with an added mechanism to provide implicit privatization. Our scheme, which we discovered independently in 2007, was also discovered by Marathe et al. (Marathe et al. 2008) who in turn attribute the idea to Detlefs et al. It works by using a simplistic GV1 global clock advanced with CAS (Dice et al. 2006) before the validation of the read-set. We also add a new *egress* global variable, whose value "chases" the clock in the manner of a ticket lock. We opted to use GV1 so we could leverage the global clock as the incoming side of a ticket lock. In the transactional load operator each thread keeps track of the most recent GV (global clock) value that it observed, and if it changed since the last load, we refresh the thread local value and revalidate the read-set. That introduces a validation cost that is in the worst case quadratic. These two changes – serializing egress from the commit – and revalidation are sufficient to give TL2 implicit privatization. These changes solve both halves of the implicit privatization problem, the 1st half being the window in commit where a thread has acquired write locks, validated its read-set, but some other transaction races past and writes to a location in the 1st thread's read-set, privatizing a region to which the 1st thread is about to write into. Serializing egress solves that problem. The 2nd half of the serialization problem is that one can end up with zombie reader transactions if a thread reads some variable and then accesses a region contingent or dependent on that variable, but some other thread stores into that variable, privatizing the region. Revalidating the read-set avoids that problem by forcing the 1st thread to discover the update and causing it to self-abort.

TLRW-IOMux A version of our read-write lock-based STM with the byte-locks replaced by a pair of counters to track read-lock acquisition. One counter is incremented upon read lock access, and the other is decremented once the read lock is released. We found this splitting of the reader-count performed better than using a single reader-count that is both incremented and decremented.

TLRW-bytelock A version of our new bytelock based TLRW algorithm that has a lock spanning a single line with $k = 48$. We also tried a lock spanning two 64 byte cache lines with $k = 112$ which we will call *TLRW-bytelock-128*. We plan to, but did not, devise a dynamic switching mechanism between the two forms though as the reader will see, the data indicates such a mechanism would be beneficial.

Our algorithm uses early (encounter order) lock acquisition and an undo write set.

TLRW-BitLock It is precisely the same as TLRW-ByteLock except that we replace the a 48-byte reader array with a 64-bit reader mask field. To keep things as similar and comparable as possible we constrained the mask field to supporting only 48 "slots," with the unslotted threads using the reader counter. Similarly, we padded the lock records so they are the same length in both TLRW-ByteLock and TLRW-BitLock. Stores of 0 or 1 into the reader array in TLRW-ByteLock code become CAS-based loops that load and set or clear the bit associated with a slotted thread. What were previously loads of a slot in the reader array now become loads of the reader bitmask and a mask/test of the thread's bit.

We begin by noting that we implemented a version of TLRW-ByteLock with lazy acquisition (instead of early acquisition and an undo write set) but do not include the results as they were not better than those yielded by TLRW-ByteLock with early acquisition. In theory lazy acquisition might yield better ultimate scalability because the write locks are held for a shorter period but lazy acquisition also requires that transactional load operators look-aside into the write-set. The look-aside overhead can be moderated through Bloom filters or through the use of a hash table to access a thread's write set, but in keeping with our goal of minimizing paths lengths for common transactional operators, we opted to use early acquisition.

Another issue we needed to resolve was to understand which fraction of the performance benefit shown by TLRW-ByteLock arises from CAS-avoidance and which fraction from the fact that we have a very efficient test to determine if a thread is already a member of the read-set for a given stripe. Not surprisingly given spatial and temporal locality it's common to find a thread read a given stripe multiple times within the same txn. That is, read-after-read is common. Without a fast thread-has-already-read-this-stripe test we'd need to revert to Bloom filters, hash tables, or simple scanning of the read-set to determine if thread was already a member of the read-set for the stripe. (If the thread was not already a reader of that stripe then we need to atomically bump the read counter and add the stripe to the thread's local read set list). Similarly, such a fast read-set membership test is also useful when upgrading a stripe from read to write status (write-after-read is also very common).

Our benchmarking showed that TLRW-bitlock exhibits awful performance when compared to TLRW-ByteLock, in particular it melted down at a concurrency level beyond 30 threads, suggesting that CAS-avoidance is the key to TLRW-bytelock performance.

Having ruled out possible benefits of these two variations of TLRW, let us move on to compare its performance with that of other the remaining algorithms listed above.

Consider the two benchmarks of Figure 1 of a Red-Black Tree with 25% puts and 25% deletes when tree size is 2K and 20K respectively, and the left side of Figure 2 when the level of modifications is down to 10%. As can be seen, the performance of TL2 and TLRW-bytelock, and TLRW-bytelock-128 are about the same, with similar scalability curves in both cases. This is encouraging since the red-black tree is a particularly trying data structure for TLRW because the transactions read sets tend to overlap at the top of the tree: in effect, the root must be locked by all transactions. As can be seen, the TLRW-bytelock slightly outperfroms the TLRW-bytelock-128 up to about 50 threads, after which the TLRW-bytelock-128 wins. This suggests that one should dynamically switch between the two, which we hope to investigate in the future.

Next, in Figure 2, we show what happens when we consider transactions with smaller overlaps. If we compare TLRW-bytelock with TL2-IP, the form of TL2 that provides implicit privatization, we can see that TLRW-bytelock has a significant performance advantage. To convince ourselves that the scalability of TLRW is due to the use of bytelocks, consider the throughput of the TLRW-IOMux algorithm. Here the same TLRW algorithm runs, with locks implemented using the best reader counters we could invent. As can be seen TLRW-IOMux performs poorly, essentially collapsing as the level of concurrency increases beyond 32 threads.

The left side of Figure 2 shows that TLRW and TL2 continue to scale about the same on a smaller tree when the level of modifications goes down, but for deferent reasons. TL2 does well, as has been explained in other papers (Dice et al. 2006) despite the high abort rate, because it locks the nodes at the head of the tree only rarely and because the cost of a retry is very low. To understand why TLRW-bytelock performs well, consider first the right-hand side of Figure 2, which shows the abort rates for the various STM implementations in the benchmark on the left of the same figure. TLRW-bytelock has significantly lower abort rates than TL2, which helps mitigate the cost of locking the head of the tree.

Next, consider Figure 3, which contains a chart that describes the common execution path (fast-path) instruction counts (assuming no concurrent activity) for transactional load and store operations in the speculative phase. In the table, *Read-after-read*, for instance, is a subsequent read to a data stripe that's already been read in the same transaction. The number V is the variable-length look-aside time where TL2 checks for a match in the write-set, and the number L is the cost of scanning the read-set for a match in TLRW-ByteLock. We note that the low costs of coherence on Sun's Niagara architecture is not unique. The new Intel $i7^{TM}$ Nahalem class X86 machines have very low storeload memory barrier and CAS costs (about 2 and 8 cycles respectively).

As noted earlier, the read sharing at the top of the red-black tree impacts TLRW performance. In Figure 4, we show what happens when we consider transactions with less read sharing. Our artificial random array benchmark, tries to capture the behavior of data structures such as hash tables that are highly distributed. In the benchmark, there is no inter-transaction locality, but within a given transaction the benchmark on the left hand side exhibits strong spatial locality (all accesses are at small offsets from the original randomly selected index) and the one on the right exhibits moderate spatial locality.

In the random array benchmark, all the TLRW algorithms outperform TL2. The TLRW-IOMux is the best performer since the cost of using CAS operations on the reader counters is low given that the sets of locations accessed are mostly disjoint and there are therefore few invalidations. Here one can also see that TLRW-bytelock which aligns along one cache line performs as well as TLRW-IOMux and outperforms TLRW-bytelock-128 that incurs

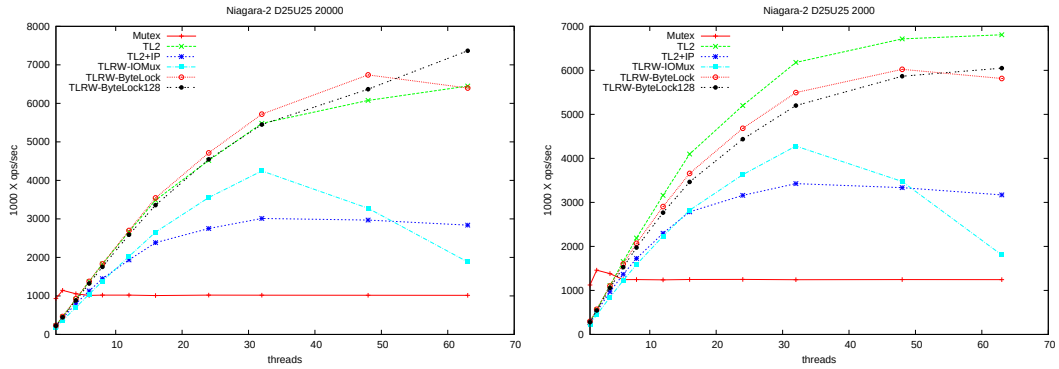

Figure 1. Throughput of Red-Black Tree with 25% puts and 25% deletes when tree size is 2K and 20K respectively on a 64 thread Niagara II.

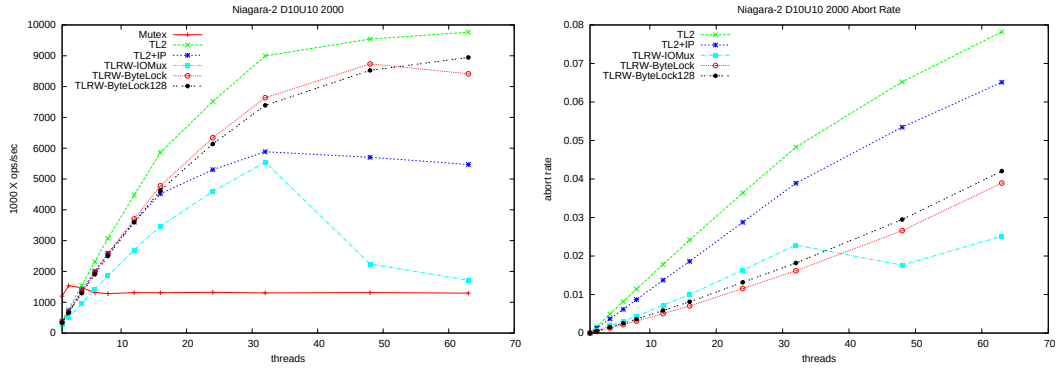

Figure 2. Throughput of Red-Black Tree with 10% puts and 10% deletes and its related abort rates (lower abort rate is better.).

Operation	Under TL2	Under TLRW-ByteLock
1st read	39 + V	24 + 1Membar
1st write	18	31 + 1CAS
Read-after-read	39 + V	12
Read-after-write	39 + V	13
write-after-read	18	39 + 1CAS + L
write-after-write	18	13

Figure 3. A chart that describes the fast-path instruction counts for loads and stores in TL2 and TLRW-bytelock transactions. Notice that we are not counting the commit time costs which are negligible for TLRW-bytelock yet involve a CAS per written location in TL2. As can be seen, TLRW-ByteLock can leverage intra-transaction spatial and temporal locality, that is, the fact that transactions re-access the same locations one after the other in the same short intervals.

an extra cache invalidation given that most locations are not shared by transactions.

Next we present the results of benchmarking a real application, the MSF (Minimum Spanning Forest) benchmark introduced by Kang and Bader (Kang and Bader 2009). The MSF program takes a graph file (we used the US Western roads system as input, just as in (Kang and Bader 2009)) and computes a minimum spanning forest. The algorithm is concurrent and the implementation by Kang and Bader uses transactional memory. A purely sequential thread-unsafe version of the program with no transactional overhead completes in 15.9 secs.

In Figure 5 we see the results of running the MSF application (The application performs a fixed amount of work and reports the duration it took). Bader and Kang reported that TL2 scaled well but the absolute performance was poor. Our results recapitulate their findings with TL2, but also show that TLRW-ByteLock both scales well and shows a significant improvement over TL2 in terms of absolute performance.

We now consider the case of irrevocable transactions. We ran a benchmark in which in addition to put and remove, we ran an iteration operator over the nodes of the tree (a classical Java library operation).

For reference, the baseline score for TLRW-byteLock without the iterator, as seen in Figure 1, is 5.5 million operations per second. In a typical run, when 31 threads executed 25% put and 25% removes, and there was one iterator thread that did not execute in irrevocable mode (i.e., it is just a normal thread) the throughput for the 31 threads dropped to 0.61 million and yet the iterator had 7718

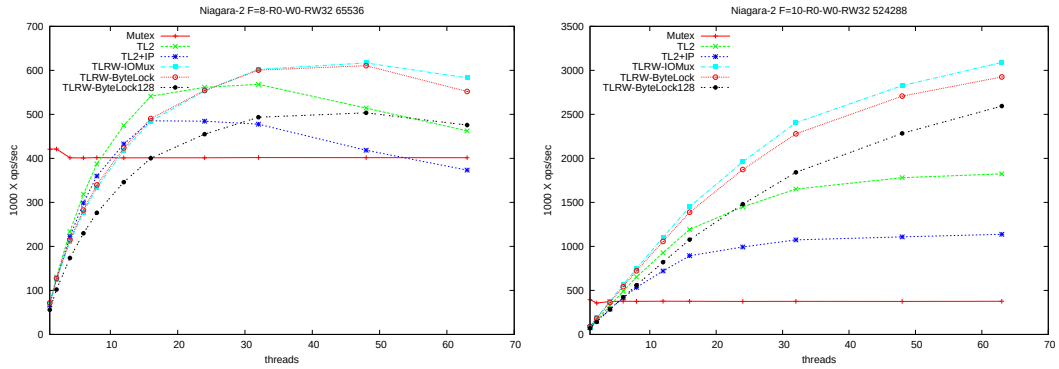

Figure 4. Throughput of the randomized work distribution benchmark on a 64 thread Niagara II. On the left a small array of 60K locations and a pattern of strong intra-transaction spatial locality and on the right 500K locations with moderate intra-transaction spatial locality. Sets of 32 locations are read and then written in these arrays.

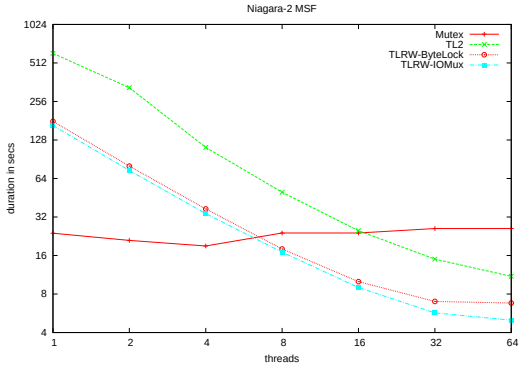

Figure 5. Latency (lower is better) of the transactified MSF application of Kang and Bader.

successes and 3990 failures. In a typical TL2 run the iterator never succeeds. This seems to support our claim that TLRW in general may have better progress properties than TL2. But those properties come at a cost because the iterator (even though it is revocable) badly degraded the performance of the other 31 threads. If we use 31 threads and the iterator thread operates in irrevocable mode, then throughput for the 31 threads drops to 0.44 million operations, and the iterator thread improves slightly to 8408 successes. However, now there are 0 failures. From our benchmarking of the irrevocable mode, we conclude that it is a good tool for guaranteeing progress (necessary in the case of I/O) but seems to have a negligible benefit for throughput.

Finally, we demonstrate how badly the TLRW algorithms perform on systems where write-sharing (coherency traffic) is expensive, which is the basis of our we claim that TLRW-bytelock should be viewed as an algorithm for single chip systems. In Figure 6 we show the throughput of a red-black tree with threads spread evenly across a 2-chip Maramba machine. The threads are not bound to cores and the operating system spreads them out so that half the threads on one chip communicate with threads on the other through an interconnect that is typically twice as slow as an on chip memory access. This proves to be an intolerable coherence cost for the TLRW algorithms. Note that if threads are restricted to one chip TLRW performs well.

4. Conclusions

This paper introduced TLRW, a new form of transactional locking that is in an algorithmic sense orthogonal to the invisible-readers based approach at the basis of all Ennals-style lock-based algorithms (Ennals). It overcomes many of the drawbacks of invisible-read based STMs, providing implicit privatization without a performance loss. The key to the new algorithm is the byte-lock, a new type of read-write lock that supports high read acquisition levels with little overhead. As our benchmarks show, TLRW using *byte-locks* suggest a new direction in STM design for the case of single chip multicore systems. Our hope is that others will find new ways to carry this approach further. Examples of possible directions are dynamically switching among the 64 and 128 array sizes, and perhaps scaling further to 3 and 4 cache lines. Also, one can think of more elaborate deadlock detection and resolution schemes, partial rollbacks of locks in a transaction, more aggressive irrevocable transaction schemes, multiplexed bytes in the bytelock instead of the reader count and so on.

Readers interested in TLRW code can email: `tlrw-feedback@sun.com`.

References

T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition edition, 2001.

T. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, University of Washington, De-

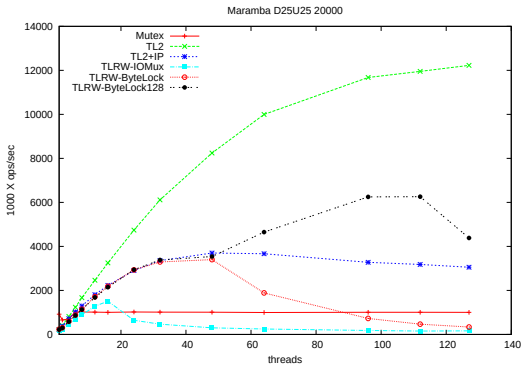

Figure 6. Throughput of Red-Black Tree on a 128 thread Maramba machine with 25% puts and 25% deletes when the tree size is 20K.

partment of Computer Science, February 1993.

D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

Dave Dice. Weblog: http://blogs.sun.com/dave/entry/cas_and_cache_trivia_invalidate, 2008.

Dave Dice and Nir Shavit. Understanding tradeoffs in software transactional memory. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 21–33, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2764-7. doi: http://dx.doi.org/10.1109/CGO.2007.38.

Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. Snzi: scalable nonzero indicators. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 13–22, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-616-5. doi: http://doi.acm.org/10.1145/1281100.1281106.

Robert Ennals. Software transactional memory should not be obstruction-free. `www.cambridge.intel-research.net/~rennals/notlockfree.pdf`. www.cambridge.intel-research.net/ rennals/notlockfree.pdf.

Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: http://doi.acm.org/10.1145/1345206.1345241.

Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: http://doi.acm.org/10.1145/1345206.1345233.

Tim Harris and Keir Fraser. Concurrent programming without locks.

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, San Mateo, CA, 2008. ISBN 0-12-370591-6.

Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM Press, 2003. ISBN 1-58113-708-7. doi: http://doi.acm.org/10.1145/872035.872048.

Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12 (3):463–492, 1990. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/78969.78972.

Seunghwa Kang and David A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, New York, NY, USA, 2009. ACM. To appear.

Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, and Dan Nussbaum Marek Olszewski. Anatomy of a scalable software transactional memory. In *Transact 2009 Workshop Submission*, 2008.

P. Magnussen, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing (IPPS)*, pages 165–171. IEEE Computer Society, April 1994.

Virendra Marathe. Personal communication. 2008.

Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of software transactional memory. Technical Report TR 893, Computer Science Department, University of Rochester, Mar 2006. Condensed version submitted for publication.

Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Scalable techniques for transparent privatization in software transactional memory. *Parallel Processing, International Conference on*, 0:67–74, 2008. ISSN 0190-3918. doi: http://doi.ieeecomputersociety.org/10.1109/ICPP.2008.69.

Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Single global lock semantics in a weakly atomic stm. In *Transact 2008 Workshop*, 2008.

Mark Moir. HybridTM: Integrating hardware and software transactional memory. Technical Report Archivist 2004-0661, Sun Microsystems Research, August 2004.

Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for c/c++. In *OOPSLA 08: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2008.

Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *20th International Symposium on Distributed Computing (DISC)*, September 2006a.

Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *TRANSACT06*, Jun 2006b.

Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM. ISBN 1-59593-189-9. doi: http://doi.acm.org/10.1145/1122971.1123001.

N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.

Michael F. Spear, Maged M. Michael, and Christoph von Praun. Ringstm: scalable transactions with a single atomic instruction. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 275–284, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-973-9. doi: http://doi.acm.org/10.1145/1378533.1378583.

Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 285–296, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-973-9. doi: http://doi.acm.org/10.1145/1378533.1378584.

Craig Zilles and Ravi Rajwar. Transactional memory and the birthday paradox. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 303–304, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-667-7. doi: http://doi.acm.org/10.1145/1248377.1248428.