Transactional Locking II

Dave Dice[1], Ori Shalev[2,1], and Nir Shavit[1]

[1] Sun Microsystems Laboratories, 1 Network Drive, Burlington MA 01803-0903,
{dice,shanir}@sun.com
[2] Tel-Aviv University, Tel-Aviv 69978, Israel,
orish@post.tau.ac.il

Abstract. The transactional memory programming paradigm is gaining momentum as the approach of choice for replacing locks in concurrent programming. This paper introduces the transactional locking II (TL2) algorithm, a software transactional memory (STM) algorithm based on a combination of commit-time locking and a novel global version-clock based validation technique. TL2 improves on state-of-the-art STMs in the following ways: (1) unlike all other STMs it fits seamlessly with any systems memory life-cycle, including those using malloc/free (2) unlike all other lock-based STMs it efficiently avoids periods of unsafe execution, that is, using its novel version-clock validation, user code is guaranteed to operate only on consistent memory states, and (3) in a sequence of high performance benchmarks, while providing these new properties, it delivered overall performance comparable to (and in many cases better than) that of all former STM algorithms, both lock-based and non-blocking. Perhaps more importantly, on various benchmarks, TL2 delivers performance that is competitive with the best hand-crafted fine-grained concurrent structures. Specifically, it is ten-fold faster than a single lock. We believe these characteristics make TL2 a viable candidate for deployment of transactional memory today, long before hardware transactional support is available.

1 Introduction

A goal of current multiprocessor software design is to introduce parallelism into software applications by allowing operations that do not conflict in accessing memory to proceed concurrently. The key tool in designing concurrent data structures has been the use of locks. Coarse-grained locking is easy to program, but unfortunately provides very poor performance because of limited parallelism. Fine-grained lock-based concurrent data structures perform exceptionally well, but designing them has long been recognized as a difficult task better left to experts. If concurrent programming is to become ubiquitous, researchers agree that alternative approaches that simplify code design and verification must be developed. This paper is interested in "mechanical" methods for transforming sequential code or coarse-grained lock-based code into concurrent code. By mechanical we mean that the transformation, whether done by hand, by a preprocessor, or by a compiler, does not require any program specific information (such

as the programmer's understanding of the data flow relationships). Moreover, we wish to focus on techniques that can be deployed to deliver reasonable performance across a wide range of systems today, yet combine easily with specialized hardware support as it becomes available.

1.1 Transactional Programming

The *transactional memory* programming paradigm of Herlihy and Moss [1] is gaining momentum as the approach of choice for replacing locks in concurrent programming. Combining sequences of concurrent operations into atomic transactions seems to promise a great reduction in the complexity of both programming and verification – by making parts of the code appear to be sequential without the need to program fine-grained locks. Transactions will hopefully remove from the programmer the burden of figuring out the interaction among concurrent operations that happen to conflict with each other. Non-conflicting Transactions will run uninterrupted in parallel, and those that do will be aborted and retried without the programmer having to worry about issues such as deadlock. There are currently proposals for hardware implementations of transactional memory (HTM) [1–4], purely software based ones, i.e. software transactional memories (STM) [5–13], and hybrid schemes (HyTM) that combine hardware and software [14, 10].[3]

The dominant trend among transactional memory designs seems to be that the transactions provided to the programmer, in either hardware or software, should be "large scale", that is, unbounded, and dynamic. *Unbounded* means that there is no limit on the number of locations accessed by the transaction. *Dynamic* (as opposed to *static*) means that the set of locations accessed by the transaction is not known in advance and is determined during its execution.

Providing large scale transactions in hardware tends to introduce large degrees of complexity into the design [1–4]. Providing them efficiently in software is a difficult task, and there seem to be numerous design parameters and approaches in the literature [5–11]. as well as requirements to combine well with hardware transactions once those become available [14, 10].

1.2 Lock-Based Software Transactional Memory

STM design has come a long way since the first STM algorithm by Shavit and Touitou [12], which provided a non-blocking implementation of static transactions (see [5–8, 15, 9–13]). A recent paper by Ennals [5] suggested that on modern operating systems deadlock prevention 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. We second this claim, noting that mechanisms already exist whereby threads might yield their quanta to other threads and that Solaris' *schedctl* allows threads to transiently defer preemption while holding locks.

[3] A broad survey of prior art can be found in [6, 15, 16].

Ennals [5] proposed an all-software lock-based implementation of software transactional memory using the object-based approach of [17]. His idea was to run through the transaction possibly operating on an inconsistent memory state, acquiring write locks as locations to be written are encountered, writing the new values in place and having pointers to an undo set that is not shared with other threads. The use of locks eliminates the need for indirection and shared transaction records as in the non-blocking STMs, it still requires however a closed memory system. Deadlocks and livelocks are dealt with using timeouts and the ability of transactions to request other transactions to abort.

Another recent paper by Saha et al. [11], uses a version of Ennals' lock-based algorithm within a run-time system. The scheme described by Saha et al. acquires locks as they are encountered, but also keeps shared undo sets to allow transactions to actively abort others.

A workshop presentation by two of the authors [18] shows that lock-based STMs tend to outperform non-blocking ones due to simpler algorithms that result in lower overheads. However, two limitations remain, limitations that must be overcome if STMs are to be commercially deployed:

Closed Memory Systems Memory used transactionally must be recyclable to be used non-transactionally and vice versa. This is relatively easy in garbage collected languages, but must also be supported in languages like C with standard malloc() and free() operations. Unfortunately, all non-blocking STM designs require closed memory systems, and the lock-based STMs [5, 11] either use closed systems or require specialized malloc() and free() operations.

Specialized Managed Runtime Environments Current efficient STMs [5, 11] require special environments capable of containing irregular effects in order to avoid unsafe behavior resulting from their operating on inconsistent states.

The TL2 algorithm presented in this paper is the first STM that overcomes both of these limitations: it works with an open memory system, essentially with any type of malloc() and free(), and it runs user code only on consistent states, eliminating the need for specialized managed runtime environments[4].

1.3 Vulnerabilities of STMs

Let us explain the above vulnerabilities in more detail. Current efficient STM implementations [18, 17, 5, 11] require closed memory systems as well as managed runtime environments capable of containing irregular effects. These closed systems and managed environments are necessary for efficient execution. Within these environments, they allow the execution of "zombies": transactions that have observed an inconsistent read-set but have yet to abort. The reliance on an accumulated read-set that is not a valid snapshot [19] of the shared memory

[4] The TL algorithm [18], a precursor of TL2, works with an open memory system but runs on inconsistent states.

locations accessed can cause unexpected behavior such as infinite loops, illegal memory accesses, and other run-time misbehavior.

The specialized runtime environment absorbs traps, converting them to transaction retries. Handling infinite loops in zombies is usually done by validating transactions while in progress. Validating the read-set on every transactional load would guarantee safety, but would also significantly impact performance. Another option is to perform periodic validations, for example, once every number of transactional loads or when looping in the user code [11]. Ennals [5] attempts to detect infinite loops by having every n-th transactional object "open" operation validate part of the accumulated read-set. Unfortunately, this policy admits infinite loops (as it is possible for a transaction to read less than n inconsistent memory locations and cause the thread to enter an infinite loop containing no subsequent transactional loads). In general, infinite loop detection mechanisms require extending the compiler or translator to insert validation checks into potential loops.

The second issue with existing STM implementations is their need for a closed memory allocation system. For type-safe garbage collected managed runtime environments such as that of the Java programming language, the collector assures that transactionally accessed memory will only be released once no references remain to the object. However, in C or C++, an object may be freed and depart the transactional space while concurrently executing threads continue to access it. The object's associated lock, if used properly, can offer a way around this problem, allowing memory to be recycled using standard malloc/free style operations. The recycled locations might still be read by a concurrent transaction, but will never be written by one.

1.4 Our New Results

This paper introduces the *transactional locking II* (TL2) algorithm. TL2 overcomes the drawbacks of all state-of-the-art lock-based algorithms, including our earlier TL algorithm [18]. The new idea in our new TL2 algorithm is to have, perhaps counter-intuitively, a global version-clock that is incremented once by each transaction that writes to memory, and is read by all transactions. We show how this shared clock can be constructed so that for all but the shortest transactions, the effects of contention are minimal. We note that the technique of time-stamping transactions is well known in the database community [20]. A global-clock based STM is also proposed by Riegel et al. [21]. Our global-clock based algorithm differs from the database work in that it is tailored to be highly efficient as required by small STM transactions as opposed to large database ones. It differs from the "snapshot isolation" algorithm of Riegel et al. as TL2 is lock-based and very simple, while Riegel et al. is non-blocking but costly as it uses time-stamps to choose between multiple concurrent copies of a transaction based on their associated execution intervals.

In TL2, all memory locations are augmented with a lock that contains a version number. Transactions start by reading the global version-clock and validating every location read against this clock. As we prove, this allows us to

guarantee at a very low cost that only consistent memory views are ever read. Writing transactions need to collect a read-set but read-only ones do not. Once read- and write-sets are collected, transactions acquire locks on locations to be written, increment the global version-clock and attempt to commit by validating the read-set. Once committed, transactions update the memory locations with the new global version-clock value and release the associated locks.

We believe TL2 is revolutionary in that it overcomes most of the safety and performance issues that have plagued high-performance lock-based STM implementations:

- Unlike all former lock-based STMs it efficiently avoids vulnerabilities related to reading inconsistent memory states, not to mention the fact that former lock-based STMs must use compiler assist or manual programmer intervention to perform validity tests in user code to try and avoid as many of these zombie behaviors as possible. The need to overcome these safety vulnerabilities will be a major factor when going from experimental algorithms to actual production quality STMs. Moreover, as Saha et al. [11] explain, validation introduced to limit the effects of these safety issues can have a significant impact on overall STM performance.
- Unlike any former STM, TL2 allows transactional memory to be recycled into non-transactional memory and back using malloc and free style operations. This is done seamlessly and with no added complexity.
- As we show in Section 3, rather encouragingly, concurrent red-black trees derived in a mechanical fashion from sequential code using the TL2 algorithm and providing the above software engineering benefits, tend to perform as well as prior algorithms, exhibiting performance comparable to that of hand-crafted fine-grained lock-based algorithms. Overall TL2 is an order of magnitude faster than sequential code made concurrent using a single lock.

In summary, TL2's superior performance together with the fact that it combines seamlessly with hardware transactions and with any system's memory life-cycle, make it an ideal candidate for multi-language deployment today, long before hardware transactional support becomes commonly available.

2 Transactional Locking II

The TL2 algorithm we describe here is a global version-clock based variant of the transactional locking algorithm of Dice and Shavit (TL) [18]. As we will explain, based on this global versioning approach, and in contrast with prior local versioning approaches, we are able to eliminate several key safety issues afflicting other lock-based STM systems and simplify the process of mechanical code transformation. In addition, the use of global versioning will hopefully improve the performance of read-only transactions.

Our TL2 algorithm is a two-phase locking scheme that employs *commit-time* lock acquisition mode like the TL algorithm, differing from *encounter-time* algorithms such as those by Ennals [5] and Saha et al. [11].

For each implemented transactional system (i.e. per application or data structure) we have a shared global version-clock variable. We describe it below using an implementation in which the counter is incremented using an increment-and-fetch implemented with a compare-and-swap (CAS) operation. Alternative implementation exist however that offer improved performance. The global version-clock will be read and incremented by each *writing transaction* and will be read by every read-only transaction.

We associate a special versioned write-lock with every transacted memory location. In its simplest form, the *versioned write-lock* is a single word spinlock that uses a CAS operation to acquire the lock and a store to release it. Since one only needs a single bit to indicate that the lock is taken, we use the rest of the lock word to hold a version number. This number is advanced by every successful lock-release. Unlike the TL algorithm or Ennals [5] and Saha et al. [11], in TL2 the new value written into each versioned write-lock location will be a property which will provide us with several performance and correctness benefits.

To implement a given data structure we allocate a collection of *versioned write-locks*. We can use various schemes for associating locks with shared data: *per object* (PO), where a lock is assigned per shared object, or *per stripe* (PS), where we allocate a separate large array of locks and memory is striped (partitioned) using some hash function to map each transactable location to a stripe. Other mappings between transactional shared variables and locks are possible. The PO scheme requires either manual or compiler-assisted automatic insertion of lock fields whereas PS can be used with unmodified data structures. PO might be implemented, for instance, by leveraging the header words of objects in the Java programming language [22, 23]. A single PS stripe-lock array may be shared and used for different TL2 data structures within a single address-space. For instance an application with two distinct TL2 red-black trees and three TL2 hash-tables could use a single PS array for all TL2 locks. As our default mapping we chose an array of 2^{20} entries of 32-bit lock words with the mapping function masking the variable address with "0x3FFFFC" and then adding in the base address of the lock array to derive the lock address.

In the following we describe the PS version of the TL2 algorithm although most of the details carry through verbatim for PO as well. We maintain thread local read- and write-sets as linked lists. Each read-set entries contains the address of the lock that "covers" the variable being read, and unlike former algorithms, does not need to contain the observed version number of the lock. The write-set entries contain the address of the variable, the value to be written to the variable, and the address of its associated lock. In many cases the lock and location address are related and so we need to keep only one of them in the read-set. The write-set is kept in chronological order to avoid write-after-write hazards.

2.1 The Basic TL2 Algorithm

We now describe how TL2 executes a sequential code fragment that was placed within a TL2 transaction. As we explain, TL2 does not require traps or the

insertion of validation tests within user code, and in this mode does not require type-stable garbage collection, working seamlessly with the memory life-cycle of languages like C and C++.

Write Transactions The following sequence of operations is performed by a writing transaction, one that performs writes to the shared memory. We will assume that a transaction is a writing transaction. If it is a *read-only transaction* this can be denoted by the programmer, determined at compile time or heuristically at runtime.

1. **Sample global version-clock:** Load the current value of the *global version clock* and store it in a thread local variable called the *read-version* number (rv). This value is later used for detection of recent changes to data fields by comparing it to the *version* fields of their versioned write-locks.
2. **Run through a speculative execution:** Execute the transaction code (load and store instructions are mechanically augmented and replaced so that speculative execution does not change the shared memory's state, hence the term "speculative".) Locally maintain a *read-set* of addresses loaded and a *write-set* address/value pairs stored. This logging functionality is implemented by augmenting loads with instructions that record the read address and replacing stores with code recording the address and value to-be-written. The transactional load first checks (using a Bloom filter [24]) to see if the load address already appears in the write-set. If so, the transactional load returns the last value written to the address. This provides the illusion of processor consistency and avoids read-after-write hazards.
   A load instruction sampling the associated lock is inserted before each original load, which is then followed by *post-validation* code checking that the location's versioned write-lock is free and has not changed. Additionally, we make sure that the lock's version field is $\leq rv$ and the lock bit is clear. If it is greater than rv it suggests that the memory location has been modified after the current thread performed step 1, and the transaction is aborted.
3. **Lock the write-set:** Acquire the locks in any convenient order using bounded spinning to avoid indefinite deadlock. In case not all of these locks are successfully acquired, the transaction fails.
4. **Increment global version-clock:** Upon successful completion of lock acquisition of all locks in the write-set perform an increment-and-fetch (using a CAS operation for example) of the global version-clock recording the returned value in a local *write-version number* variable wv.
5. **Validate the read-set:** validate for each location in the read-set that the version number associated with the versioned-write-lock is $\leq rv$. We also verify that these memory locations have not been locked by other threads. In case the validation fails, the transaction is aborted. By re-validating the read-set, we guarantee that its memory locations have not been modified while steps 3 and 4 were being executed. In the special case where $rv + 1 = wv$ it is not necessary to validate the read-set, as it is guaranteed that no concurrently executing transaction could have modified it.

6. **Commit and release the locks:** For each location in the write-set, store to the location the new value from the write-set and release the locations lock by setting the version value to the write-version wv and clearing the write-lock bit (this is done using a simple store).

A few things to note. The write-locks have been held for a brief time when attempting to commit the transaction. This helps improve performance under high contention. The Bloom filter allows us to determine if a value is not in the write-set and need not be searched for by reading the single filter word. Though locks could have been acquired in ascending address order to avoid deadlock, we found that sorting the addresses in the write-set was not worth the effort.

Low-Cost Read-Only Transactions One of the goals of the proposed methodology's design is an efficient execution of *read-only transactions*, as they dominate usage patterns in many applications. To execute a read-only transaction:

1. **Sample the global version-clock:** Load the current value of the *global version-clock* and store it in a local variable called *read-version* (rv).
2. **Run through a speculative execution:** Execute the transaction code. Each load instruction is *post-validated* by checking that the location's versioned write-lock is free and making sure that the lock's version field is $\leq rv$. If it is greater than rv the transaction is aborted, otherwise commits.

As can be seen, the read-only implementation is highly efficient because it does not construct or validate a read-set. Detection of read-only behavior can be done at the level of of each specific transaction site (e.g., method or atomic block). This can be done at compile time or by simply running all methods first as read-only, and upon detecting the first transactional write, abort and set a flag to indicate that this method should henceforth be executed in write mode.

2.2 A Low Contention Global Version-Clock Implementation

There are various ways in which one could implement the global version-clock used in the algorithm. The key difficulty with the global clock implementation is that it may introduce increased contention and costly cache coherent sharing. One approach to reducing this overhead is based on splitting the global version-clock variable so it includes a version number and a thread id. Based on this split, a thread will not need to change the version number if it is different than the version number it used when it last wrote. In such a case all it will need to do is write its own version number in any given memory location. This can lead to an overall reduction by a factor of n in the number of version clock increments.

1. Each version number will include the thread id of the thread that last modified it.

2. Each thread, when performing the load/CAS to increment the global version-clock, checks after the load to see if the global version-clock differs from the thread's previous wv (note that if it fails on the CAS and retries the load/CAS then it knows the number was changed). If it differs, then the thread does not perform the CAS, and writes the version number it loaded and its id into all locations it modifies. If the global version number has not changed, the thread must CAS a new global version number greater by one and its id into the global version and use this in each location.
3. To read, a thread loads the global version-clock, and any location with a version number $> rv$ or $= rv$ and having an id different than that of the transaction who last changed the global version will cause a transaction failure.

This has the potential to cut the number of CAS operations on the global version-clock by a linear factor. It does however introduce the possibility of "false positive" failures. In the simple global version-clock which is always incremented, a read of some location that saw, say, value $v + n$, would not fail on things less than $v + n$, but with the new scheme, it could be that threads 1..n-1 all perform non-modifying increments by changing only the id part of a version-clock, leaving the value unchanged at v, and the reader also reads v for the version-clock (instead of $v + n$ as he would have in the regular scheme). It can thus fail on account of each of the writes even though in the regular scheme it would have seen most of them with values $v...v + n - 1$.

2.3 Mixed Transactional and Non-Transactional Memory Management

The current implementation of TL2 views memory as being clearly divided into transactional and non-transactional (heap) space where mixed-mode transactional and non-transactional accesses are proscribed. As long as a memory location can be accessed by transactional load or store operations it must not be accessible to non-transactional load and store operations and vice versa. We do however wish to allow memory recycled from one space to be reusable in the other. For type-safe garbage collected managed runtime environments such as that of the Java programming language, any of the TL2 lock-mapping policies (PS or PO) provide this property as the GC assures that memory will only be released once no references remain to an object. However, in languages such as C or C++ that provide the programmer with explicit memory management operations such as malloc and free, we must take care never to free objects while they are accessible. The pitfalls of finding a solution for such languages are explained in detail in [18].

There is a simple solution for the *per-stripe* (PS) variation of TL2 (and in the the earlier TL [18] scheme) that works with any malloc/free or similar style pair of operations. In the transactional space, a thread executing a transaction can only reach an object by following a sequence of references that are included in the transaction's read-set. By validating the transaction before writing the

locations we can make sure that the read set is consistent, guaranteeing that the object is accessible and has not been reclaimed. Transacted memory locations are modified after the transaction is validated and before their associated locks are released. This leaves a short period in which the objects in the transaction's write-set must not be freed. To prevent objects from being freed in that period, threads let objects *quiesce* before freeing them. By *quiescing* we mean letting any activity on the transactional locations complete by making sure that all locks on an object's associated memory locations are released by their owners before. Once an object is quiesced it can be freed. This scheme works because any transaction that may acquire the lock and reach the disconnected location will fail its read-set validation.

Unfortunately, we have not found an efficient scheme for using the PO mode of TL2 in C or C++ because locks reside inside the object header, and the act of acquiring a lock cannot guaranteed to take place while the object is alive. As can be seen in the performance section, on the benchmarks/machine we tested there is a penalty, though not an unbearable one, for using PS mode instead of PO.

In STMs that use encounter-time lock acquisition and undo-logs [5, 11] it is significantly harder to protect objects from being modified after they are reclaimed, as memory locations are modified one at a time, replacing old values with the new values written by the transaction. Even with quiescing, to protect from illegal memory modifications, one would have to repeatedly validate the entire transaction before updating each location in the write-set. This *repeated* validation is inefficient in its simplest form and complex (if at all possible) if one attempts to use the compiler to eliminate unnecessary validations.

2.4 Mechanical Transformation of Sequential Code

As we discussed earlier, the algorithm we describe can be added to code in a mechanical fashion, that is, without understanding anything about how the code works or what the program itself does. In our benchmarks, we performed the transformation by hand. We do however believe that it may be feasible to automate this process and allow a compiler to perform the transformation given a few rather simple limitations on the code structure within a transaction.

We note that hand-crafted data structures can always have an advantage over TL2, as TL2 has no way of knowing that prior loads executed within a transaction might no longer have any bearing on results produced by transaction.

2.5 Software-Hardware Interoperability

Though we have described TL2 as a software based scheme, it can be made inter-operable with HTM systems. On a machine supporting dynamic hardware transactions, transactions need only verify for each location read or written that the associated versioned write-lock is free. There is no need for the hardware transaction to store an intermediate locked state into the lock word(s). For every write they also need to update the version number of the associated lock

upon completion. This suffices to provide interoperability between hardware and software transactions. Any software read will detect concurrent modifications of locations by a hardware writes because the version number of the associated lock will have changed. Any hardware transaction will fail if a concurrent software transaction is holding the lock to write. Software transactions attempting to write will also fail in acquiring a lock on a location since lock acquisition is done using an atomic hardware synchronization operation (such as CAS or a single location transaction) which will fail if the version number of the location was modified by the hardware transaction.

3 Empirical Performance Evaluation

We present here a set of microbenchmarks that have become standard in the community [25], comparing a sequential red-black tree made concurrent using various algorithms representing state-of-the-art non-blocking [6] and lock-based [5, 18] STMs. For lack of space we can only present the red-black tree data structure and only four performance graphs.

The sequential red-black tree made concurrent using our transactional locking algorithm was derived from the `java.util.TreeMap` implementation found in the Java programming language JDK 6.0. That implementation was written by Doug Lea and Josh Bloch. In turn, parts of the Java TreeMap were derived from the Cormen et al. [26]. We would have preferred to use the exact Fraser-Harris red-black tree [6] but that code was written to their specific transactional interface and could not readily be converted to a simple form.

The sequential 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 insert a new element describing the key-value pair. If the key is already present in the data structure put will simply update 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 describes the maximum possible size (capacity) of the data structure.

For our experiments we used a 16-processor Sun Fire™ V890 which is a cache coherent multiprocessor with 1.35Ghz UltraSPARC-IV® processors running Solaris™ 10. As claimed in the introduction, modern operating systems handle locking well, even when the number of threads is larger than the number of CPUs. In our benchmarks, our of STMs used the Solaris *schedctl* mechanism to allow threads to request short-term preemption deferral by storing to a thread-specific location which is read by the kernel-level scheduler. Preemption

deferral is advisory - the kernel will try to avoid preempting a thread that has requested deferral. We note that unfortunately we could not introduce the use of schedctl-based preemption deferral into the hand crafted lock-based *hanke* code, the lock-based *stm_ennals* code described below, or *stm_fraser*. This affected their relative performance beyond 16 threads but not in the range below 16 threads.

Our benchmarked algorithms included:

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

stm_fraser This is the state-of-the-art non-blocking STM of Harris and Fraser [6]. We use the name originally given to the program by its authors. It has a special record per object with a pointer to a transaction record. The transformation of sequential to transactional code is not mechanical: the programmer specifies when objects are transactionally opened and closed to improve performance.

stm_ennals This is the lock-based encounter-time object-based STM algorithm of Ennals taken from [5] and provided in LibLTX [6]. Note that LibLTX includes the original Fraser and Harris lockfree-lib package. It uses a lock per object and a non-mechanical object-based interface of [6]. Though we did not have access to code for the Saha et al. algorithm [11], we believe the Ennals algorithm to be a good representative this class of algorithms, with the possible benefit that the Ennals structures were written using the non-mechanical object-based interface of [6] and because unlike Saha et al., Ennals write-set is not shared among threads.

TL/PO A version of our algorithm [18] which does not use a global version clock, instead it collects read and write-sets and validates the read-set after acquiring the locks on the memory locations. Unlike TL2, it thus requires a safe running environment. We bring here the per-object locking variation of the TL algorithm.

hanke This is the hand-crafted lock-based concurrent relaxed red-black tree implementation of Hanke [27] as coded by Fraser [6]. The idea of relaxed balancing is to uncouple the re-balancing from the updating in order to speed up the update operations and to allow a high degree of concurrency.

TL2 Our new transactional locking algorithm. We use the notation TL2/PO and TL2/PS to denote the per-object and per-stripe variations. The PO variation consistently performed better than PS, but PS is compatible with open memory systems.

In Figure 1 we present four red-black tree benchmarks performed using two different key ranges and two set operation distributions. The key range of $[100, 200]$ generates a small size tree while the range $[10000, 20000]$ creates a larger tree, imposing larger transaction size for the set operations. The different operation distributions represent two type of workloads, one dominated by reads (5% puts, 5% deletes, and 90% gets) and the other (30% puts, 30% deletes, and 40% gets) dominated by writes.

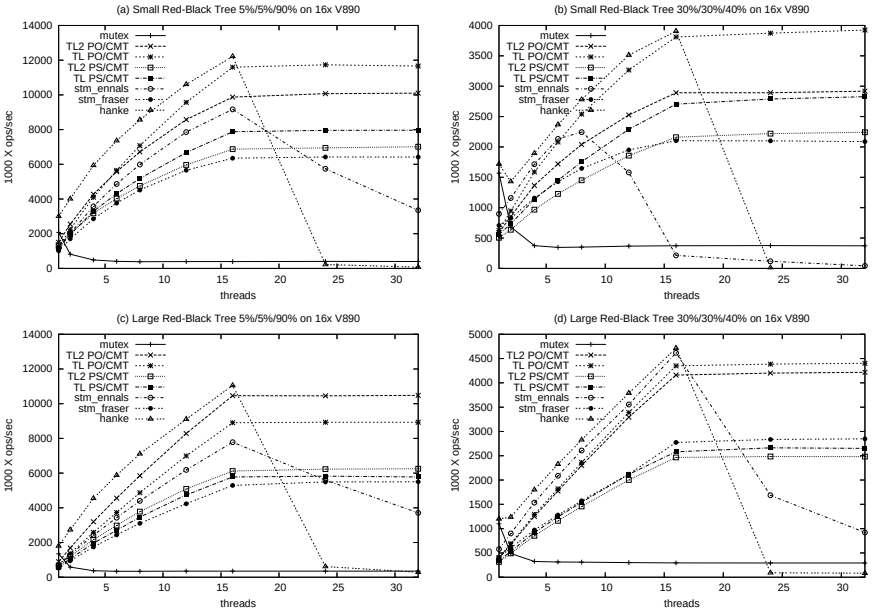

Fig. 1. Throughput of Red-Black Tree with 5% puts and 5% deletes and 30% puts, 30% deletes

In all four graphs, all algorithms scale quite well to 16 processors, with the exception of the mutual exclusion based one. Ennals's algorithm performs badly on the contended write-dominated benchmark, apparently suffering from frequent transaction collisions, which are more likely to occur in encounter-time locking based solutions. Beyond 16 threads, the Hanke and Ennals algorithms deteriorate because we could not introduce the *schedctl* mechanism to allow threads to request short-term preemption deferral. It is interesting to note that the Fraser-Harris STM continues to perform well beyond 16 threads even without this mechanism because it is non-blocking. As expected, object based algorithms (PO) do better than stripe-based (PS) ones because of the improved locality in accessing the locks and the data.

The performance of all the STM implementations usually differs by a constant factor, most of which we associate with the overheads of the algorithmic mechanisms employed (as seen in the single thread performance). The hand-crafted algorithm of Hanke provides the highest throughput in most cases because its single thread performance (a measure of overhead) is superior to all STM algorithms. On the smaller data structures TL/PO (or TL/PS) performs better than TL2/PO (respectively TL2/PS) because of lower overheads, part of which can be associated with invalidation traffic caused by updates of the version clock (this is not traffic caused by CAS operations on the shared location. It is due to the fact that the location is being updated). This is reversed and TL2 becomes

superior when the data structure is large because the read-set is larger and read-only transactions incur less overhead in TL2. The TL and TL2 algorithms are in most cases superior to Ennals's STM and Fraser and Harris's STM. In all benchmarks they are an order of magnitude faster than the single lock Mutex implementation.

4 Conclusion

The TL2 algorithm presented in this paper provides a safe and easy to integrate STM implementation with reasonable performance, providing a programming environment similar to that attained using global locks, but with a ten-fold improvement in performance. TL2 will easily combine with hardware transactional mechanisms once these become available. It provides a strong indication that we should continue to devise lock-based STMs.

There is however still much work to be done to improve TL2's performance. A lot of these improvements may require hardware support, for example, in implementing the global version clock and in speeding up the collection of the read-set. The full TL2 code will be publicly available shortly.

References

1. Herlihy, M., Moss, E.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the Twentieth Annual International Symposium on Computer Architecture. (1993)
2. Rajwar, R., Herlihy, M., Lai, K.: Virtualizing transactional memory. In: ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture, Washington, DC, USA, IEEE Computer Society (2005) 494–505
3. Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S.: Unbounded transactional memory. In: HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, Washington, DC, USA, IEEE Computer Society (2005) 316–327
4. Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional memory coherence and consistency. In: ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture, Washington, DC, USA, IEEE Computer Society (2004) 102
5. Ennals, R.: Software transactional memory should not be obstruction-free. www.cambridge.intel-research.net/ rennals/notlockfree.pdf (2005)
6. Harris, T., Fraser, K.: Concurrent programming without locks. www.cl.cam.ac.uk/Research/SRG/netos/papers/2004-cpwl-submission.pdf (2004)
7. Herlihy, M.: The SXM software package. `http://www.cs.brown.edu/~mph/SXM/README.doc` (2005)
8. Herlihy, M., Luchangco, V., Moir, M., Scherer, W.: Software transactional memory for dynamic data structures. In: Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing. (2003)
9. Marathe, V.J., Scherer III, W.N., Scott, M.L.: Adaptive software transactional memory. In: Proceedings of the 19th International Symposium on Distributed Computing, Cracow, Poland (2005)

10. Moir, M.: HybridTM: Integrating hardware and software transactional memory. Technical Report Archivist 2004-0661, Sun Microsystems Research (2004)
11. Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., Hertzberg, B.: A high performance software transactional memory system for a multi-core runtime. In: To appear in PPoPP 2006. (2006)
12. Shavit, N., Touitou, D.: Software transactional memory. Distributed Computing **10**(2) (1997) 99–116
13. Welc, A., Jagannathan, S., Hosking, A.L.: Transactional monitors for concurrent objects. In: Proceedings of the European Conference on Object-Oriented Programming. Volume 3086 of Lecture Notes in Computer Science., Springer-Verlag (2004) 519–542
14. Ananian, C.S., Rinard, M.: Efficient software transactions for object-oriented languages. In: Proceedings of Synchronization and Concurrency in Object-Oriented Languages (SCOOL), ACM (2005)
15. Marathe, V.J., Scherer, W.N., Scott, M.L.: Design tradeoffs in modern software transactional memory systems. In: LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems, New York, NY, USA, ACM Press (2004) 1–7
16. Rajwar, R., Hill, M.: Transactional memory online. `http://www.cs.wisc.edu/trans-memory` (2006)
17. Harris, T., Fraser, K.: Language support for lightweight transactions. SIGPLAN Not. **38**(11) (2003) 388–402
18. Dice, D., Shavit, N.: What really makes transactions fast? In: TRANSACT06 ACM Workshop. (2006)
19. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM **40**(4) (1993) 873–890
20. Thomasian, A.: Concurrency control: methods, performance, and analysis. ACM Comput. Surv. **30**(1) (1998) 70–119
21. Riegel, T., Felber, P., Fetzer, C.: A Lazy Snapshot Algorithm with Eager Validation. In: 20th International Symposium on Distributed Computing (DISC). (2006)
22. Agesen, O., Detlefs, D., Garthwaite, A., Knippel, R., Ramakrishna, Y.S., White, D.: An efficient meta-lock for implementing ubiquitous synchronization. ACM SIGPLAN Notices **34**(10) (1999) 207–222
23. Dice, D.: Implementing fast java monitors with relaxed-locks. In: Java Virtual Machine Research and Technology Symposium, USENIX (2001) 79–90
24. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7) (1970) 422–426
25. Herlihy, M., Luchangco, V., Moir, M., Scherer, III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the twenty-second annual symposium on Principles of distributed computing, ACM Press (2003) 92–101
26. Cormen, T.H., Leiserson, Charles, E., Rivest, R.L.: Introduction to Algorithms. MIT Press (1990) COR th 01:1 1.Ex.
27. Hanke, S.: The performance of concurrent red-black tree algorithms. Lecture Notes in Computer Science **1668** (1999) 286–300