Towards a Fully Pessimistic STM Model

Alexander Matveev
Tel-Aviv University
matveeva@post.tau.ac.il

Nir Shavit
MIT and Tel-Aviv University
shanir@csail.mit.edu

Abstract

The designs of software transactional memory (STM) algorithms to date have been optimistic: transactions that run into inconsistencies abort and retry. The common view is that this optimistic approach gives significant performance benefits, and yet we know that it also results in complex programming, limitations on what can be executed within a transaction, and difficult debugging. This is a burden that does not exist in the pessimistic lock-based programming model transactions are meant to replace.

This paper introduces the first STM system that is fully pessimistic, that is, each and every transaction, whether reading or writing, is executed once and never aborts. The benefits of this fully pessimistic STM are that programming with it is logically as simple as with locks, allowing I/O and system calls within a transaction, and making the debugging process significantly simpler. Perhaps surprisingly, we show that on many standard STM benchmarks, our fully pessimistic STM system, which also offers full transactional privatization, delivers performance and scalability that are comparable to that of the most efficient optimistic non-privatizing STM systems. This puts in question our commonly accepted understanding of the tradeoffs between pessimism and performance.

1. Introduction

Transactional memory systems are one of the leading approaches to overcoming the difficulties of lock-based programming, with an expected debut in GCC 4.7 of a commercial quality software transactional memory (STM) system. All STM algorithms (see [10]) to date, including the TinySTM algorithm of Felber, Fetzer, and Reigel [9] and the TL2 STM of Dice, Shalev, and Shavit [6], are optimistic or partially optimistic: some transactions can run into inconsistencies and be forced to abort and retry. The common view is that this optimistic approach gives significant performance benefits when compared to pessimistic transactions, and there has been some lower bound work to support this claim [4].

The use of optimistic transactions however, introduces various limitations into the programming model, ones that do not exist in the lock-based algorithms they set out to replace. Because a transaction can fail and will need to be retried, one is restricted when performing external system based operations such as I/O calls; if you read from the disk and fail, what do you do with the data? Similarly, special STM friendly versions of operations such as malloc and free must be designed if we wish them to be executed within a transaction. This is because they need to be ready for the possibility of an abort of the transaction immediately after allocating or freeing. This limitation complicates programming even further if a transaction calls code from other sources, code that may contain I/O or malloc operations written by other programmers. The possibility of aborts also complicates debugging. Debuggers go through the code step by step. If at any point the code can be rolled back (as a result of an abort) depending on the parallel execution, the simple step by step logic cannot be applied. The bug can be revealed only when some specific "back jumps" are performed. Therefore, simply debugging in the usual way will not help.

There have been several attempts to rectify this situation. Welc et al. [21] introduced the notion of irrevocable transactions. They provided the first STM system providing pessimistic transactions that can be executed concurrently with optimistic ones. Their system can execute one pessimistic transaction at a time, and use this pessimistic transaction to perform I/O or other externally dependent operations. This system was the first to answer the need to execute systems calls within transactions, but does not relieve the programmer from the having to plan and be aware of which operations to run within the specialized pessimistic transaction. It still leaves the same limitations on all other optimistic transactions, even if they are read-only transactions.

Perelman et al. [14] show a partially pessimistic STM that can support read-only transactions that do not abort. Their STM does so by keeping multiple versions of the transactions' view during its execution. Though theoretically interesting, the overhead of maintaining and checking these multiple versions makes this STM impractical. Attiya and Hillel [3] presented the first partially pessimistic STM that provides read-only transactions without multiple versions. However, their solution requires acquiring a read-lock for every location being read. This overhead, again, is too high to make this algorithm practical when compared to state-of-the-art STMs such as TinySTM and TL2, in which reading a location does not require any form of locking, or for that matter, any form of writing to shared memory.

In this paper we introduce the first STM system that is fully pessimistic: each and every transaction, whether reading or writing, is executed once and never aborts. The benefits of this fully pessimistic STM are that unlike all previous optimistic or partly pessimistic STMs, in which the programmer has to worry about the effects of failed and repeated code, here "what you write is what you get." It allows actions like I/O and system calls to be used freely within transactions, relieves the programer of the need to understand the internals of imported libraries and imported code, and makes the task of debugging much simpler. We believe this makes the job of transactifying legacy sequential or lock-based code (a major undertaking of today's concurrent programmers) significantly simpler.

But what is the implementation overhead of this fully pessimistic STM? Our algorithm executes write transactions sequentially in a manner similar to [21], yet allows concurrent read-only

transactions without using read-locks or multiple versions as in [3, 14]. We do so by using a TL2/LSA [6, 15] style time-stamping scheme together with a new variation on the quiescence array mechanism of Matveev and Shavit [1]. The sequential execution of the pessimistic write transactions is a drawback relative to standard TL2, but also has some helpful performance advantages. The most important one is that our STM transactions do not acquire or release locks using relatively expensive CAS operations. Because there is only one write transaction at a time, we notice that one can update locations and their timestamps using a sequence of simple stores followed by a single memory barrier. Moreover, one does not need read-location logging and revalidation or any bookkeeping for rollback in the case of aborts. Our use of the Matveev and Shavit quiescence mechanism is a variation on the mechanism, which was originally used to provide privatization of transactions, in order to allow writes to track concurrent read-only transactions with little overhead. The writes avoid writing if it can interrupt reads, and thus allows reads to execute without aborting. A side benefit of this mechanism is that our new fully pessimistic STM also provides implicit privatization with very little overhead (Achieving implicit privatization efficiently in an STM is not an easy task and has been the subject of a flurry of recent research [1, 2, 11–13, 18, 19]).

Perhaps surprisingly, our fully pessimistic and privatizing STM provides performance comparable to that of the fully optimistic non-privatizing state-of-the-art TL2 algorithm on a wide variety of accepted STAMP benchmarks [5] and red-black trees. How could this be? As we show, the key observation is that in many real-world applications, transactions are not executed immediately one after the other. Rather, there is typically a gap of time, between transactional calls, in which applications perform useful non-transactional work. Our STM makes use of this time to complete sequentially executing non-aborting write transactions and to perform the notification and synchronization operations that would in other STMs fall directly in the computation's critical path.

Obviously, there are cases where the gaps are small, or worst yet, that the application has very high levels of transactional writes (such as in STAMP's Labyrinth benchmark). In such cases our above implementation will deliver poor performance. To overcome some of this problem, we propose to combine pessimistic STMs with lightweight transactional threads [8] which we will call fibers (we believe this is a name used historically for such lightweight user-level threads). We will allow each hardware thread to run multiple transactions as fibers. The fibers monitor the progress of write transactions and whenever a write transaction is queued, waiting to be executed, the fiber will switch to another fiber to possibly allow a read transaction to execute concurrently on the same hardware thread. As we show in the context of a red-black tree benchmark (unfortunately introducing fibering into STAMP is an undertaking beyond the scope of this paper), this approach provides a significant enhancement of our STM's performance under high levels of transactional writes to the tree.

In summary, this paper is not claiming that we should replace optimistic STMs with pessimistic ones. We are however providing the first ever fully pessimistic privatizing STM system, and arguing that there are many cases (and even more cases when combined with fibering) in which the pessimistic STM's significantly enhanced user experience is provided with no performance penalty. This brings to question our accepted ideas about the performance characteristics of STMs, and down the road may help to improve their design.

2. Fully Pessimistic Transactions

Our design of an STM in which all transactions are pessimistic, that is, do not abort, will build on prior work of Welc et al. [21], which shows how to allow one pessimistic write transaction (the authors call it irrevocable) to execute in parallel with multiple optimistic ones. A typical transaction must read and write to multiple locations. If a transaction involves only reads we call it a *read transaction*, and otherwise it is a *write transaction*. The pessimistic transaction in [21] uses special mark-bits to prevent other optimistic transactions from writing to locations the pessimistic transaction is reading, and has the pessimistic transaction wait for the release of any locks it encounters on the locations it is writing. Because of this need to wait, one can only run a single irrevocable write transaction at a time; otherwise we would run into a situation in which we either have deadlock (both transactions waiting for each other to release locks) or some transaction will have to abort. In fact, this deadlock situation is unavoidable, and one cannot in general run more than a single pessimistic write transaction at a time in any STM system. Our algorithm here will extend the idea in Welc et al. [21] of running only one pessimistic write transaction at a time, but unlike Welc et. al, we will support multiple concurrently executing pessimistic read transactions. The STM will also provide full privatization [20].

The sequential execution of the pessimistic write transactions is a drawback relative to standard STM's such as TL2, but also has some advantages. One key idea in the new algorithm is that we can get away with transactions not acquiring or releasing locks (which requires relatively expensive CAS operations, and is a major source of overhead in traditional STMs). Because there is only one write transaction at a time, we show how one can write to locations and update their timestamps using a sequence of simple stores followed by a single memory barrier. Moreover, we show that one does not need read-location logging and revalidation, or bookkeeping for aborts. We will elaborate on this below.

Another key idea is to efficiently pause reads and writes if they might affect each other, and by doing so save on the overhead of read transactions' operations having to be visible. We do so by revising the quiescence mechanism of Matveev and Shavit [1], an algorithmic technique that was originally developed to provide transactional privatization. This mechanism allows read transactions to complete without aborting, and to do so with very low overhead.

2.1 The New Fully Pessimistic STM

We now show in detail how the two mechanisms can be combined to create our new fully pessimistic STM system.

Our STM will execute all write transactions one after another. Writes will be stored to a local write-set and written to the shared memory in the commit phase. Read transactions will be executed in parallel with writes and other read transactions. The idea at the core of the implementation is to prevent any write transaction from writing on locations which are currently being read by some concurrent read transaction. New values to be written will be stored in a local write-set which is not written until the commit. In the commit phase, the write-locations will be blocked from being read by any new read transactions, and a quiescence pass will be executed to make sure the write-locations are not being accessed by concurrent readers. Only then will the writer be allowed to write the locations from the write-set to the memory. As we can prove, this combination of blocking and delaying will guarantee that all read and write transactions are internally and externally consistent (i.e. serializable).

The first element of our solution will be to use a version-number-based consistency mechanism in the style of the TL2 algorithm of Dice, Shalev, and Shavit [6]. The range of shared memory is divided to stripes, and with each we associate a local version-number (in a similar way to [6, 7, 9]). For example, if the shared memory range is 2^{20} bytes and a stripe is defined to be 2^8 bytes, then there will be a total of 2^{12} stripes and associated version-numbers.

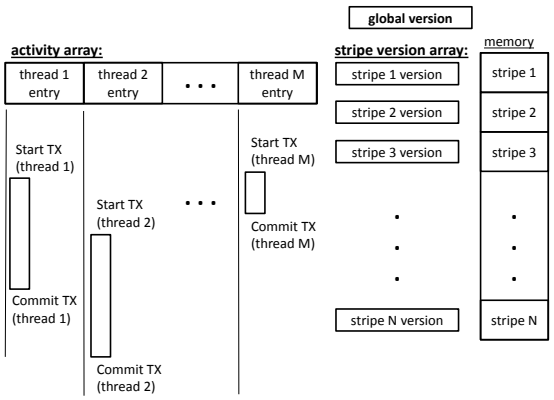

Figure 1. Global variables of the algorithm

In addition, there will be a shared global version number (as introduced by [6, 16]), a 64bit unsigned integer initialized to 1. For any transaction we will denote the set of the memory locations read as the *read-set* and the set of the locations written as the *write-set*. The key idea will be, as in TL2, to determine a location's validity by checking that its version number relative to the global version number. The actual test however will differ from the one in TL2.

Our *quiescence mechanism* will be a variation of the mechanism we used in [1]. The quiescence mechanism uses a global *activity array*. For now lets assume that this array has an entry per thread in the system, indexed by its ID. We will later show how to reduce the array size to be a function of the number of hardware cores. The entry for a specific thread indicates if the thread is currently executing a transaction. It consists of a **tx_version** field, a 64bit value that is initialized to current global version upon transaction start (default value is the maximum 64bit value 0xFF..FF).

Figure 1 depicts the global variables: the division of the shared memory to stripes with associated stripes' versions, the global version number used for consistency, and the *activity_array* used for inter-thread coordination.

The synchronization of the read and write transactions is performed by blocking the read operations and delaying the write of the write-set locations. The global version number is used to implement this blocking and delaying. Every transaction, upon starting, will read the current global version number to the *tx_version* variable associated with the transaction's thread. A read operation within a transaction will block when the *tx_version* is equal to the read location's version number. The blocking is implemented by a *Wait for version progress* procedure that reads the global version repeatedly until it is not equal to the transaction's *tx_version*. On loop exit, the transaction's *tx_version* is updated to the new global version read.

When the system starts, the stripes' version numbers are initialized to 0 and the global version number is initialized to 1. The tricky part of the algorithm is the synchronization between the commit phase of the write transaction and the concurrent transactions' read operations. The commit phase of the write transaction has to accomplish two main tasks:

1. **Memory update barrier:** write the write-set locations to memory so that no abort occurs.
2. **Signal next writer:** allow the next write transaction to execute.

First, we explain the implementation of (1) and then explain how to implement (2).

The *memory update barrier* of the commit is composed of the following steps: (1) set every write-set location's version to *tx_version + 1*, (2) add 1 to the global version, (3) quiescence, (4) write of the write-set to memory, and (5) add 1 to the global version.

Assume we have a write transaction that arrived at the commit phase. It begins by updating the write-set locations' version numbers to the *tx_version + 1* and then increments the global version number by 1. Any new concurrent transactions started after the global version increment step will be blocked from reading a write location that is in the write-set of the concurrently committing write transaction. This is because the write location's version will be equal to the concurrent transaction's *tx_version*. Next, the quiescence will be performed by the committing write transaction to ensure that all concurrently executing transactions that may read the write-set locations have finished: the *activity array* will be scanned for entries having a *tx_version* less than current global version. The writing transaction will spin on every entry for which that condition is true and until it becomes false. Once it completes this quiescing phase, no other concurrent transactions can read the write-set locations and therefore the transaction can write them safely to memory. Finally, the actual writes to memory are performed and the global version number is incremented again to release the possible blocked concurrent transactions (ones spinning on locations in the write set).

After a transaction read operation is blocked and released, it cannot be blocked again. In other words, the *Wait for version progress* loop can be initiated only once. This is because the blocking can be performed only for transactions that were started during the commit phase of the concurrent writer. Therefore, if a transaction was blocked and released, then the concurrent writer must have committed. The next write transaction cannot block transactions that started before it. This means that the read operations' stripe version validations are not required after a blocking was executed once. As a result, a local variable *progress_is_seen* is used within the transaction to indicate that a *Wait for version progress* has been already executed. Initially, it is set to FALSE and upon completion of the blocking it is set to TRUE.

As we noted, we assume for now that a thread with id *th_id* is associated with *activity_array[th_id]* entry. A read transaction executed by such a thread will be implemented in the following way:

Upon read transaction start:

1. **Read global version:** Read the global version to *tx_version* variable in the *activity_array[th_id]*.
2. **Execute a memory fence**.

For every read location of the read transaction:

1. **Check progress indication:** Tests if the *progress_is_seen* is TRUE. If so, jump to the last step.
2. **Check location version:** Tests if the location's version number is not equal to the *tx_version*. If so, jump to the last step.
3. **Wait for version progress:** Execute the *Wait for version progress* procedure that spins until the global version it is not equal to the *tx_version*. Set the transaction local variable *progress_is_seen* to TRUE.
4. **Read the memory:** Read the memory location value and return.

Upon commit of the read transaction:

1. **Set inactivity:** Set the *tx_version* variable in the *activity_array[th_id]* to the maximum value.

We now introduce the *Signal next writer* component of the commit. In [21] the write transactions coordination is implemented by a global *writer_lock*. Every write transaction tries to acquire this global lock on start and release it on finish. This lock acquire

and release sequences can cause high cache coherence traffic. To avoid this, we implement a different scheme using a combination of a *writer_lock* and a simple "pass the baton" style *signaling mechanism* in the activity array.

We add to each entry of the activity array a *writer_waiting* boolean field. Upon start of a write transaction, the *writer_waiting* field in the *activity array* entry will be set to TRUE and then the global *writer_lock* will be checked to see if it is taken. If free, a lock will attempt to acquire the lock. Otherwise, the lock is already taken, and the write transaction will wait for a signal by spinning on its *writer_waiting* field until it becomes FALSE. Upon commit of currently executing write transaction, the activity array will be scanned for entries with *writer_waiting* set to TRUE. These entries represent the concurrent write transactions which are waiting for a signal. One writer will be chosen and signaled by setting the writer's entry *writer_waiting* flag to FALSE. If no such entries found, the *writer_lock* will be released.

In the common case, there is some degree of concurrency between the write transactions. Therefore, usually during the commit of a write transaction there will be some entry in the activity array with *writer_waiting* set to TRUE. As a result, the global lock acquire and release operations will be infrequent and most of the time write transactions will signal each other, invalidating only one cache line in a specific core.

When a write transaction signals the next writer to execute, it must choose between currently waiting writers. A policy must be defined to avoid starvation of some thread. Suppose currently committing writer has a thread id of *writer_th_id*. To make the system fair we scan the activity array for an entry with a waiting writer starting from *writer_th_id + 1* to the array end and then from 0 to *writer_th_id* (not included). In this way every waiting writer will be signaled after at most a number of time proportional to the array size.

A simple way to add the signaling is by executing it after the commit is done. This will make the read operations of the writer very simple - just look at the local write-set and if the location is not there read from the memory. In general we want to signal the next writer as soon as possible because of the writers serial bottleneck. The earliest point for signal execution is after the write-set locations are blocked for new transactions - after first global version increment. In this point the next write transaction can start it's execution. If it will try to read a blocked location - it will wait until the currently executing commit is finished. When it will arrive to it's own commit a check will be performed for a concurrent commit still running and it will wait for it to finish. All this waitings will be implemented using the same *Wait for version progress* procedure described before.

A write transaction executed by thread with id *th_id* will be implemented in the following way:

Upon start of the write transaction:

1. **Set writer activity:** Set the thread's entry *writer_waiting* in the *activity_array[th_id]* to TRUE.
2. **Execute a memory fence**
3. **Writer Gate 1 (Common Case) - Check for signal:** Test if *writer_waiting* has become FALSE. If so continue to the *Read global version* step.
4. **Writer Gate 2 (Rare Case) - Check for writer lock:** Test if *writer_lock* is free. If so, try to acquire it with a CAS. If success, set *writer_waiting* to FALSE and continue to the *Read global version* step.
5. **Go to check signal step:** Jump back to step 2.
6. **Read global version:** Read the global version to *tx_version* variable in the *activity_array[th_id]*
7. **Execute a memory fence**

For every read location of the write transaction:

1. **Check local write-buffer:** Check if the read location is in the write-buffer already. If so, then return the value stored there.
2. **Check progress indication:** Test if the *progress_is_seen* is TRUE. If so, jump to the last step.
3. **Check location version:** Test if the location's version number is not equal to the *tx_version*. If so, jump to the last step.
4. **Wait for version progress:** Execute the *Wait for version progress* procedure. Then set the transaction local variable *progress_is_seen* to TRUE.
5. **Read the memory:** Read the memory location value and return.

For every write location of the write transaction:

1. **Update the local write-buffer:** Write the pair (address, value) to the local write-buffer.

Upon commit of the write transaction:

1. **Sync with the concurrent writer:** Check if *tx_version* is even. If so, execute the *Wait for version progress* procedure. (This step is necessary to ensure that only one write transaction performs the commit at a time. Also note, that on loop exit the *tx_version* will be updated to the current global version)
2. **Update the write-set versions:** Write *tx_version + 1* to every write-location version in the write-set.
3. **First global version increment:** Increment the global version by 1 and update the *tx_version* to the new global version number. (this effectively locks readers from reading write-set locations).
4. **Execute a memory fence**
5. **Signal next writer:** Scan the *activity array* for an entry with *writer_waiting* equal to TRUE. If found, set the entry's *writer_waiting* to FALSE. Otherwise, release the *writer_lock*. To prevent starvation the scan is performed from *th_id + 1* to the end of the activity array and then from *0* to the *thread_id + 1*. (The thread ids are not operating system ids; They are internal ids used to index directly the activity array).
6. **Quiescence:** Scan the activity array for entries with *tx_version* less than *current global version* (or *tx_version*). For every such entry spin until this condition becomes false. (The meaning of this is to wait for all transactions that have been started before the *First global version increment* step of this commit phase. Transactions started after this step will be blocked upon trying to read the write locations of this commit).
7. **Write the write-set:** Write all the write-locations to the memory.
8. **Execute a memory fence**
9. **Second global version increment:** Increment the global version by 1 and update the *tx_version* to the new global version number.

To illustrate the synchronization between the write and read transactions, Figure 2 shows 3 stages of a concurrent execution. In stage 1, there is read transaction 1 and write transaction 1. The read transaction 1 started before the writer so it proceeds without being blocked. The writer reads the stripes and writes to the local write-set. In stage 2, in order for the write transaction to commit, the write-set locations versions (stripes 1 and 2) are updated to *tx_version + 1*$= 24$ and then the first global version increment is

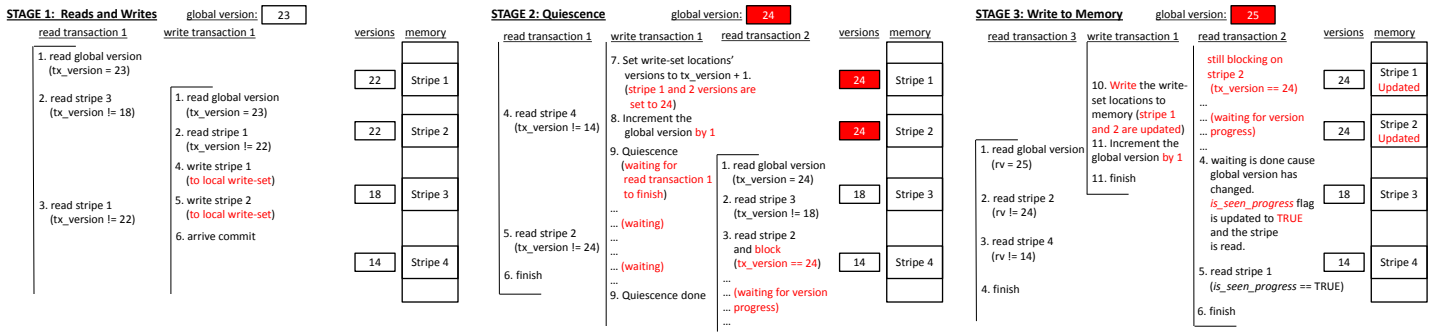

Figure 2. Three different stages of concurrent execution between read and write transactions are shown.

performed (to 24) (step 7 and 8). This blocks the write locations from being read by new transactions started after step 8. Therefore, a newly started read transaction 2 is blocked when it tries to read stripe 2. The write transaction 1 continues to the quiescence (step 9) and finds read transaction 1 with a *tx_version* that is less than the current global version. The writer waits for this reader because it started before the blocking step of the commit (step 8). When the read transaction 1 finishes, the quiescence takes place and the writer performs the writing to memory in stage 3. Then the global version is incremented for a second time and the spinning read transaction 2 is unblocked. Transaction 2 sees that the global version has advanced and updates its *progress_is_seen* flag to TRUE. From this point it is not required to validate versions because it cannot be blocked any more.

The key to this algorithm's performance is that the above pessimistic transaction implementation is, in a weird way, more efficient than standard optimistic one as in, say, TL2 or TinySTM; all of the above operations, and in particular operations in the commit-phase, do not acquire locks using expensive hardware CAS operations. Instead, the "locking" of all the write-set locations is done by updating their version numbers and incrementing the global version by 1. All are simple hardware store operations, and the global version increment is followed by a single memory fence. Moreover, because transactions are sequential, one does not need read-location logging and revalidation, saving a lot of booking overhead. Finally, the relevant bookkeeping for aborts and back jump tricks are not required.

For lack of space we do not prove the new STM correctness and progress guarantees. Informally, the commits can be executed one at a time, so the quiescence steps can be executed, also, one at a time. The commit's *first global version increment* step splits all of the transactions executed to two disjoint groups; the (1) ones started before this step, and the (2) ones started after it. The quiescence step waits for the transactions of the first group, while the transactions of the second will be blocked by reading the current write transaction's write-set location. Therefore, after quiescence complete, the write-set locations are "sterilized" from possible concurrent reads, and the write-set can be flushed to the shared memory safely. After the write is done, the *first global version increment* step unblocks the (possibly blocked) transactions from the second group.

2.2 Privatization

One of the great benefits of the new fully pessimistic algorithm is that we get implicit privatization almost for free. A common and useful programming pattern is to isolate a memory segment accessed by some thread, with the intent of making it inaccessible to other threads. This "privatizes" the memory segment, allowing the owner access to it without having to use the costly transactional protocol. (for example, a transaction could unlink a node from a transactionally maintained concurrent list in order to operate on it more efficiently in a non-transactional manner.)

Why is guaranteeing implicit privatization such a problem? Consider a transaction that has just privatized a memory segment. Even though the segment cannot be accessed by any other transaction (executing on the same or other processor) after the transaction commits, prior to the commit, latent transactional loads and stores might be pending. These latent loads and stores, executed by transactions that accessed the segment before it was isolated, can still read from and write into the shared memory segment that was intended to be isolated. This unexpected behavior is known as the "privatization problem." This results in unexpected changes to the contents of the isolated shared memory (which may have been reallocated and (although resident in the shared memory)) is intended to be outside of the transactionally shared data region. Other unexpected, generally asynchronous, behaviors can also occur.

Figure 3 shows an example of such a scenario. In modern STM algorithms such as TinySTM and TL2, in order to be efficient, read only transactions are invisible, that is, they do not write (let along hold lock) to memory locations they read, which prevents a concurrent writer from knowing they might be reading the location. Consider a write transaction by a thread P that removes a node from a linked list. Once the transaction completes, the node will no longer be reachable to other threads and P will be able to operate on it privately. However, before P completes its transaction, another transaction Q reads the pointer, and is poised to read a value from the node. P has no way of detecting Q. This is because Q reads the pointer invisibly, and will not see any trace of P touching the location in the node since P is operating on it non-transactionally. As a result, even though Q is doomed to fail (once it revalidates the locations it read, and detects the pointer has changed), in the interim it can perform illegal operations. This is an example of a privatization problem that one must overcome.

In general, in order to privatize some shared memory range, two steps must be taken:

1. *Disable access by new transactions to this range.*
2. *Wait for transactions already accessing the range to complete.*

In the example in Figure 3, in order to provide privatization as described in the two steps, a transaction needs to remove the node from the list and then wait for other transactions which might concurrently read the node to finish. After both steps are done the node is privatized - there are no transactions currently accessing it.

For lack of space we do not prove the new STM provides privatization. Informally, the first step of removing the node from the list is the programmer's responsibility. The more problematic

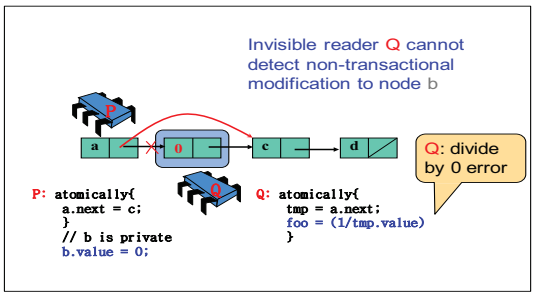

Figure 3. Privatization Pathology Example.

is the second one. The fully pessimistic STM achieves the second one without any changes because the write transaction's commit-phase performs the second step within its quiescence step. The quiescence step waits for currently executing read transactions to finish. Because this step is executed after the write-locations are blocked for new transactions, and because and it waits for the currently executing ones to finish - it accomplishes the both of the required steps towards privatization in a straightforward way.

2.3 Transactions and Fibering

The algorithm executes the write transactions one after another. This can lead to a large scalability penalty when there are many writers executing at the same time. A possible solution for this is a context-switch. Kernel context switches are relatively expensive operations that cannot be executed frequently. Therefore, we propose to implement userspace switching using fibers (or coroutines). There are existing packages [8] which implement userspace switching. Our implementation does not use any existing packages, it only simulates the userspace switching in the most simple way by a thread function call when the switch is required. We believe this is enough to show the performance benefits if fibering were to be applied.

To explain the implementation we start with an example. Suppose we have a some physical thread and userspace A and B. The physical thread starts to execute thread A and now encounters a writer transaction that is forced to wait for a writer token. Instead of waiting, thread A performs a userspace context switch to thread B. In thread B, after every read transaction, a check is performed to see if the writer token has been received. When received, execution is returned to thread A and the pending write transaction is executed. Otherwise, thread B continues its execution until it encounters a write transaction which is also forced to wait. In this case, there is no further possibility to switch (because we have only A and B) and thread B waits for the write token. When it arrives, thread B's write transaction is executed. Then execution is return to thread A, the pending writer is executed, and thread A continues its execution normally.

In our implementation, a userspace thread is identified by a *start_function*. Groups of userspace threads are assigned to physical threads. Physical thread starts by executing the first userspace thread from his group. If a wait for a writer token is encountered, an internal switch to the second userspace thread in the group is performed by calling the second thread's start function. The second thread will check if the writer token is received for the physical thread after every read transaction and if received, the execution will return back to the calling thread's function. Upon a write transaction if a wait for the writer token is required, an internal switch to the third userspace thread in the group is performed by calling third thread's function. The wait for the write token will be performed in a spin-loop when the pool of the userspace threads for this physical thread is finished. In this way, the userspace threads in the group are called recursively, on a wait condition, until the condition is satisfied, and then "collapse" the execution backwards.

The use of fibering provides our implementation with two benefits:

1. **Reduced writer contention:** Because the userspace switch is done at the contention point, when waiting for the writer token is required, a chance is given to some other thread to execute code which is not a write transaction. This saves the idle cycles to allow other work to go on in the system.
2. **Bounds the *activity array* length:** If we have N userspace threads, and M hardware threads, then we can assign a group of N/M userspace threads to every hardware thread. This means that one needs only M *activity array* entries to implement the required quiescence mechanism.

As we will show in the next section, the combination of fibers and pessimistic transactions enhances performance.

3. Empirical Performance Evaluation

We empirically evaluated our algorithm on an Oracle SPARC Niagara 2 based 128-way machine and an Intel Core i7 8-way machine.

The algorithms we benchmarked are:

TL2 The transactional locking algorithm of [6]. This algorithm is representative of a class of high performance lock-based algorithms such as [9, 17, 21].

PTM Our new fully pessimistic STM. We use the initials PTM in the graphs, though this is not its "name."

FiberedPTM The same algorithm as PTM but with fibering implemented and applied after peek performance is reached. The threads are divided to groups of two. When some thread's write transaction is required to wait for a signal, a user space context switch is performed to pass execution to the next thread in the thread's group.

The fully pessimistic algorithm design allows read transactions to be executed concurrently with write transactions. But because only one write transaction is allowed be executed at any time, we lose the benefit of writer transactions concurrency which optimistic STMs have. On the other hand, the new write transactions are more efficient and never abort. In addition, there can be an unlimited number of concurrent read-only transactions. We have thus designed two benchmarks that will allow us to evaluate the effects of these parameters: a red-black tree implementation and a collection of STAMP micro-benchmarks. We will use the the standard STAMP benchmarks to gauge the possible effect of the seriality of the writes and the overhead of our synchronization mechanism if they were to be executed in real applications. We will use the red-black tree to perform instrumentations that we can vary artificially in order to understand how various parameters (such as the gap between one transaction execution and the next) affect performance.

3.1 The Red Black Tree Benchmark

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 present, else updates the key's node value. *Delete* removes the key's node, if present, and *get* returns the value associated with a key. We allow the tree to grow to maximum 200K elements while initially making it 100K elements. The execution is done with 10% puts and 10% deletes when we vary the length of the private work which is done after the writer transaction. *N private work* means threads execute N opcodes of dummy memory fence to simulate the private gaps between the transactions.

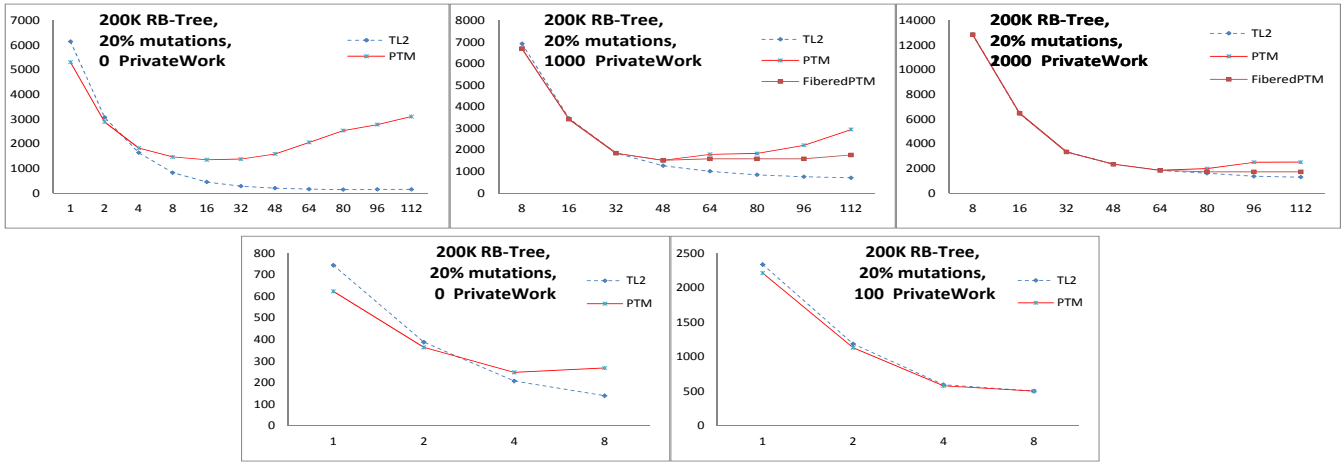

Figure 4. Latency of 200K sized Red-Black Tree with 10% puts and 10% deletes. The top graphs are Oracle results, for different private work amounts: 0, 1000, and 2000 opcodes. The bottom graphs are Intel results, for 0 and 100 opcodes of private work.

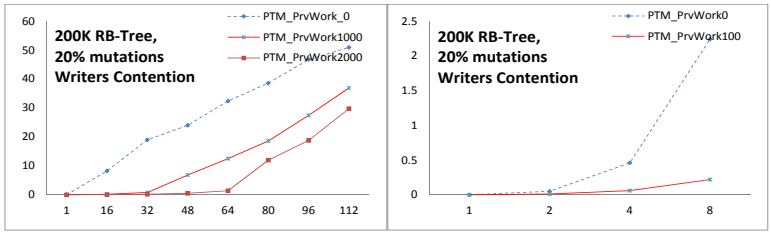

Figure 5. Write transactions contention indicators for the RB-Tree benchmarks on Oracle and Intel.

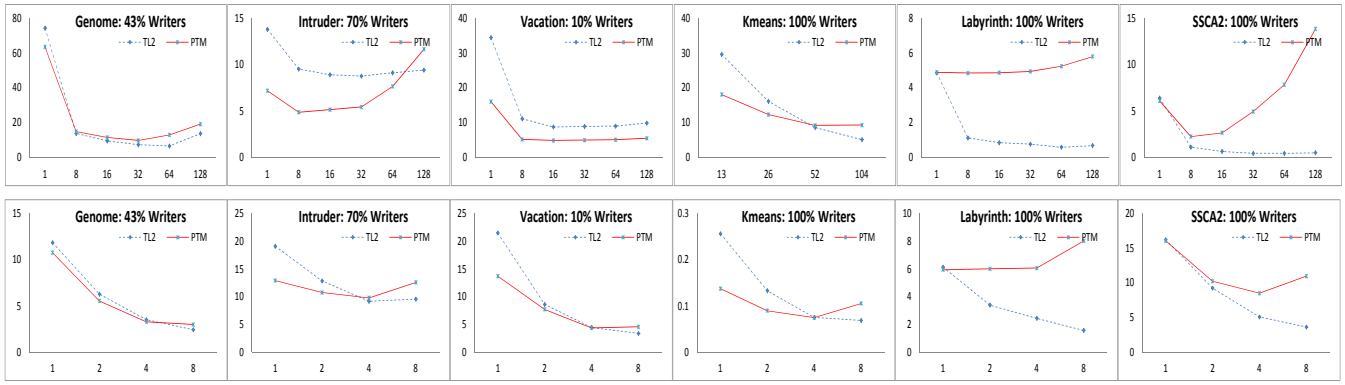

Figure 6. The top graphs show STAMP benchmark results on Oracle and the bottom graphs show the results on Intel.

Figure 4 presents the results for different gaps, that is, amounts of "private work" executed after the write transactions. For the Oracle machine, when the amount of private work is 0, which is the case when transactions are executed one after another without any gaps - PTM scales only up to 4 threads. This result is not surprising because this is the case when the serial bottleneck of the write transactions dominates the performance. In the case of 1000 opcodes of private work, the PTM scales up to 48 threads and then drops in the performance. By applying the fibering after 48, threads the performance drop is avoided and is constant after this point. For the 2000 opcode private work case, the PTM scales up to 64 threads. By applying fibering after 64 threads, the algorithm scales up to 80 threads and then sustains the performance. On Intel, when executing for 0 private work, PTM scales till 4 threads and then drops. This is similar to the Oracle case. Executing for 100 opcodes of private work gives the same performance and slightly better than of TL2. Interestingly, 100 opcodes of private work are required to achieve the same performance in the Intel case.

The results for the RB-Tree benchmark are good on Oracle and Intel CPUs when private work is introduced after the write transaction. This private work's effect is to reduce write transaction frequency, the main problem for the fully pessimistic STM perfor-

mance. To analyze this more deeply we measured the write transaction contention for RB-Tree benchmarks. The writers contention is defined by the average number of write transactions waiting in the activity array when a the currently executing write transaction must choose the next writer by scanning this array. In figure 5, write transaction contention results are presented for both Oracle and Intel CPU for the benchmarks of figure 4. PTM-PrvWork0, PTM-PrvWork1000 and PTM-PrvWork2000 are the writers contention results for 0, 1000 and 2000 opcodes of private work on Oracle. When the writers contention gets closer to the number 10 (10 write transactions waiting on average) the PTM starts to lose to TL2. This indicates that the fully pessimistic STM can sustain up to 10 write transactions which are serialized and still be better than the optimistic STM. For Intel, PTM-PrvWork0 and PTM-PrvWork100 are the contention results for 0 and 100 opcodes of private work. In this case, when the contention level 0.5 is reached, the PTM loses to TL2. An 0.5 level is much less than the Oracle result which is 10. The reason for this is that Intel is significantly more pipelined and optimized for single-thread performance than Oracle.

The RB-Tree results analysis show that the PTM performance depends on (1) write transactions contention, and (2) write transaction single-thread performance. By improving the single-thread performance at the peek performance point (when there is no more scaling) we can extend the performance to a higher number of threads. On Oracle the PTM write transaction is much more efficient than on Intel. In contrast, on Oracle the contention is much higher with a 128-way machine relative to the Intel 8-way machine. Still the total combination of the writers contention with a single-thread performance a writer give us good results on both processors when the private work between transactions is sufficiently large.

Finally, as we see in Figure 4, fibering can be used to preserve or improve the performance for increasing number of threads. This is a result of better CPU utilization, when hardware threads are executing write transactions that are waiting for one another.

3.2 STAMP Benchmarks

The STAMP benchmarks [5] model realistic programs. We implemented the fully pessimistic STM into STAMP and tested it relative to the TL2 optimistic STM. The fully pessimistic model requires from the user to indicate which transaction is a read-only. In STAMP there is no such indication. Therefore we analyzed every benchmark for read transactions and added the read-only indication in appropriate transactions. The benchmarks we implemented are: *genome*, *intruder*, *vacation*, *kmeans*, *labyrinth* and *ssca2*.

In figure 6, the STAMP results are shown for Oracle and Intel machines. Genome, Intruder and Vacation show that the fully pessimistic STM performs well when there is some combination of read and write transactions. It performs well even for a very high number of cpu which may be surprising. They have some private work between the transactions. In the Intruder case, PTM wins by almost a factor of two. That's because the write transaction of PTM is faster than that of TL2 because unlike TL2, it never aborts (while TL2 suffers from a high abort rate) and it commits using only writes and no CAS operations. On Intel, the same three benchmarks have similar behavior except Intruder. PTM loses to TL2 beyond 4 threads. This is because on Intel, the contention is much lower so the aborts have a lesser affect and the Intel cpu is able more optimized and pipelined so the effects of the simpler transaction code are limited.

Kmeans is a benchmark which has 100% write transactions. Still, we can see that PTM performs pretty well on both Oracle and Intel. The reason PTM performs well despite its serialized write transaction sequences, is that there are big gaps of private between transactions. In contrast, in Labyrinth and SSCA2, benchmarks which have also a 100% write transactions, PTM performs poorly on both Oracle and Intel. PTM loses on both because the write transactions do not have a long enough private gaps between them to may the serial execution of the write transactions.

In summary, our results indicate that the main reasons for the good performance of the fully pessimistic STM are

- **Infrequent writers:** The chance of having many write transactions executing is low in most STAMP benchmarks. Is this true of large scale real-world applications? We do not know but it would be interesting to program one using our new STM and find out, and
- **Efficient pessimistic writers:** The pessimistic write transaction is more efficient than the optimistic one. No read-set maintenance and revalidation is required at any point. In addition, the commit-phase does not include any CAS operations: only simple assignments.

References

[1] D. D. A. Matveev and N. Shavit. Implicit privatization using private transactions. In *Transact 2010*, Paris, France, 2010.

[2] H. Attiya and E. Hillel. The cost of privatization. In *DISC*, pages 35–49, 2010.

[3] H. Attiya and E. Hillel. Single-version stms can be multi-version permissive. In *Proceedings of the 12th international conference on Distributed computing and networking*, ICDCN'11, pages 83–94, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 3-642-17678-X, 978-3-642-17678-4. URL `http://portal.acm.org/citation.cfm?id=1946143.1946151`.

[4] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. *Theory of Computing Systems*, pages 1–22, 2010-12-01. URL `http://dx.doi.org/10.1007/s00224-010-9304-5`.

[5] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[6] 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.

[7] A. Dragojevic, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 155–165, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: http://doi.acm.org/10.1145/1542476.1542494.

[8] R. S. Engelschall. Gnu pth - the gnu portable threads, 2006. URL `http://www.gnu.org/software/pth/`.

[9] P. Felber, C. Fetzer, and T. 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.

[10] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010. ISBN 1608452352, 9781608452354.

[11] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *2009, 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT09*, 2009.

[12] H. Machens and V. Turau. Avoiding Publication and Privatization Problems on Software Transactional Memory. In N. Luttenberger and H. Peters, editors, *17th GI/ITG Conference on Communication in Distributed Systems (KiVS 2011)*, volume 17 of *OpenAccess Series in Informatics (OASIcs)*, pages 97–108, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-27-9. doi: http://dx.doi.org/10.4230/OASIcs.KiVS.2011.97. URL `http://drops.dagstuhl.de/opus/volltexte/2011/2961`.

[13] V. J. Marathe, M. F. Spear, and M. 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.

[14] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in stm. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 16–25, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-888-9. doi: http://doi.acm.org/10.1145/1835698.1835704. URL `http://doi.acm.org/10.1145/1835698.1835704`.

[15] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *20th International Symposium on Distributed Computing (DISC)*, September 2006.

[16] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *In Proceedings of the 20th International Symposium on Distributed Computing (DISC06*, pages 284–298, 2006.

[17] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. 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.

[18] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in stm. *SIGPLAN Not.*, 42:78–88, June 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1273442.1250744. URL `http://doi.acm.org/10.1145/1273442.1250744`.

[19] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 338–339, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-616-5. doi: http://doi.acm.org/10.1145/1281100.1281161. URL `http://doi.acm.org/10.1145/1281100.1281161`.

[20] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 34–48, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2764-7. doi: http://dx.doi.org/10.1109/CGO.2007.4. URL `http://dx.doi.org/10.1109/CGO.2007.4`.

[21] A. Welc, B. Saha, and A.-R. 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.