Towards an efficient STM that ensures local progress

João Pereira
NOVA LINCS - Departamento de Informática
FCT - Universidade Nova de Lisboa
jmd.pereira@campus.fct.unl.pt

Abstract. Software Transactional Memory (STM) systems should ensure two fundamental properties: safety and liveness. Opacity and local progress are, respectively, the strongest safety and liveness properties that are ensured by these systems. It has been proven that in faulty environments, these properties can’t be ensured together. With this work we aim at designing an STM system that can satisfy both opacity and local progress for fault-free environments.

Keywords: software transactional memory, concurrency control

1 Introduction

Transactional Memory (TM) attempts at simplifying parallel programming by establishing a new concurrency control mechanism. Based on the model used in database systems, transactions are used to manage memory accesses by concurrent threads.

A transaction is a sequence of operations that appears to be atomic: its effects must either be observed in their entirety, or not observed at all. Also, transactions should be serializable: their effects must appear to take place at a single point in time and in a consistent order. This means that a situation in which several transactions run simultaneously should be indistinguishable from one in which they execute one after the other.

By using transactions, writing parallel programs becomes simpler: a programmer can wrap a set of instructions that should be atomic inside a transaction, instead of having to use low-level mechanisms such as locks and semaphores to specify how to ensure atomicity. The transactional memory system becomes the one responsible for ensuring atomicity by managing the concurrent transactions. TM systems can be implemented using hardware and/or software constructs. This work focuses on Software Transactional Memory (STM).

STM systems can be implemented in many ways. But regardless of the design choices made, these systems should be able to ensure two properties: safety and liveness. Most lock-based STM systems already ensure opacity [5] as a robust safety property. There are three liveness properties [1], each one stronger than the previous:
Solo progress: every transaction which eventually runs alone is guaranteed to commit.

Global progress: at least one transaction is always guaranteed to commit.

Local progress: every transaction is guaranteed to commit.

It has been shown [1] that STMs cannot satisfy opacity and local progress simultaneously in the presence of faults, i.e. transactions that never attempt to commit due to a crash or an infinite loop. The goal of this ongoing work is to design a lock-based STM that ensures opacity and local progress in the absence of faults.

We start with a system that already ensures opacity and solo progress, and modify it by employing mechanisms that will make it able to ensure local progress. We chose the TinySTM [3] implementation to serve as a basis for our work.

2 Related Work

In this section we describe existing proposals on how to achieve opacity and local progress. Our work will later be introduced with this context in mind.

2.1 Providing opacity using a global clock

Informally, the opacity property says that every transaction must observe a consistent state of the system, i.e., a state produced by a sequence of previously committed transactions.

A popular way to achieve opacity is using a lock-based design with a global clock. This strategy is used by the TL2 algorithm [2].

TL2 is a word-based STM, which means that its basic locking units are memory words, as opposed to objects in the case of object-based systems. Each memory address is mapped by a hash function to a position in a big array of versioned locks, which combine a mutual-exclusion lock with a version number used by conflict detection.

TinySTM's algorithm resembles TL2. Below we describe generally the TL2 algorithm.

1. When a transaction starts, it samples the global clock, and stores its current value in a thread-local variable we'll call \( tx\text{-version} \).
2. While the transaction is running:
   - When a load (read) occurs, the versioned-lock associated with the address to be loaded is checked. If its version is greater than \( tx\text{-version} \), that means the memory location has been modified after the current transaction has started. Since the transaction is not seeing a consistent view of memory, it is aborted (i.e., restarted with a new \( tx\text{-version} \)). After a successful load, the address and its version are stored in the transaction's read-set, so that the previous inconsistency can be checked for all read addresses in a final commit-time validation. Note that with
Towards an efficient STM that ensures local progress

this strategy the reader does not leave any trace of its operation in shared memory: this strategy is called *invisible reads* and is desirable for scalability. The opposite strategy is *visible reads*, meaning that a read operation leaves behind some sort of metadata that signals that the read occurred, e.g., the reader could increment a “number of active readers” counter.

- **When a store (write) occurs**, the address and value to be written are stored in the transaction’s *write-set*, and the value is only actually stored later at commit-time, after validation has occurred. This strategy is called *lazy updates*. The opposite strategy is *eager updates*, where the write occurs immediately on shared memory, but it requires that the associated versioned-lock be acquired immediately as well.

3. **When the transaction commits:**
   
   (a) **All the locks in the write-set are acquired.** If some lock is already owned by another transaction, the current one aborts.
   
   (b) **The global clock is incremented.** An atomic increment-and-fetch is performed, and the returned value is stored in a *write-version* variable.
   
   (c) **The read-set is validated.** For each location in the read-set, its version number is checked. If some location has \( \text{version} > \text{tx-version} \), it means that while the transaction was running, that memory location was modified, and the transaction aborts. Otherwise if all reads were valid the transaction can commit, since it has seen a consistent view of memory.
   
   (d) **The transaction is committed.** The write-set values are effectively stored in memory. Their versions are set to *write-version*, and their locks are released.

2.2 Providing local progress

The local progress property says that every transaction must eventually commit. Below are three ways to achieve this. Table 1 summarizes the properties of these approaches, along with our proposed solution.

**Global lock.** The simplest way of providing local progress is making transactions acquire a single, program-wide, fair mutual exclusion lock, so that they run one at a time without conflicts. However, since this option leaves out any concurrency, we discard it.

**Priorities.** An acceptable solution is to use the greedy contention manager [4]. When a transaction begins, it is given a timestamp which it retains even if it aborts and restarts. A transaction with an earlier timestamp has higher priority than a transaction with a later timestamp. If a transaction performs an access that conflicts with a higher priority transaction, it aborts. This mechanism is able to provide local progress, but only when coupled with visible reads. This last requirement has the disadvantage of adding latency to the reads of every transaction because they need to write to shared memory, which in turn can induce contention and limit scalability.
Irrevocability. A final approach is to use irrevocable transactions. These transactions do not abort, and are guaranteed to commit. Only one irrevocable transaction can be active at a time. Local progress is achieved by turning systematically aborted transactions into irrevocable ones, assuring their progress.

Some care has to be taken in implementing irrevocability: when an irrevocable transaction is running in parallel with one or more speculative transactions, if the irrevocable transaction reads a variable, no speculative transaction can commit a write to that variable, as that would invalidate the irrevocable transaction’s read-set, forcing it to abort (a contradiction). There are two ways to circumvent this.

1. **Serial irrevocability** [7]: When an irrevocable transaction starts it acquires a global lock which makes all speculative transactions suspend their commit. Invisible reads can be maintained for irrevocable transactions, but parallelism is sacrificed as speculative transactions may need to wait for the privileged transaction to finish to be able to commit.

2. **Using read locks**: Irrevocable transactions acquire read locks on read operations. When a speculative transaction tries to write on a read-locked location, it either aborts or spins, so that the irrevocable transaction’s read is not invalidated. Concurrency is preserved with this solution, however, read locks must be released one by one at commit time, increasing the commit complexity.

Table 1: Properties of the approaches to achieve local progress. $R$ and $W$ denote the transaction’s read and write set, respectively. Our approach achieves the best of serial and read-lock irrevocability: it allows concurrent execution and committing with $O(1)$ commit complexity for privileged (irrevocable) transactions.
Table 1 summarizes the properties of each approach to achieve local progress. Global lock precludes any concurrency. Priorities requires all transactions to use visible reads. With serial irrevervability, irrevocable transactions use invisible reads at the expense of preventing speculative transactions from committing concurrently with the execution of an irrevocable transaction. Note that serial irrevervable transactions have a constant commit complexity because they only need to release their lock so that speculative transactions can resume their commits. Read-lock irrevervability makes the opposite trade-off: read-lock irrevocable transactions now use visible reads by acquiring read locks, but speculative transactions can safely commit concurrently. Unfortunately with this approach the commit complexity of read-lock irrevocable transactions is no longer constant.

In this paper we propose an alternative approach that combines the best of both serial and read-lock irrevervability: speculative transactions can commit concurrently with irrevocable transactions while these enjoy a commit with constant complexity.

3 Our Solution

Our solution is based on irrevervability. Local progress is ensured using the concept of privileged transactions. A privileged transaction is a transaction that never aborts, and there can only be one privileged transaction active at a time. Although by this definition a privileged transaction is an irrevocable transaction, we call them privileged instead of irrevocable to disambiguate between our algorithm and the algorithm with $O(|R| + |W|)$ commit complexity [8, 9].

Our algorithm captures the best properties of both serial and read-lock irrevervability: it allows $O(1)$ commit complexity for privileged transactions, and concurrent commits.

3.1 Design Choices

In order to achieve $O(1)$ commit complexity for privileged transactions, eager updates must be used, i.e. on store operations values are written directly to memory, as opposed to the lazy updates approach in which values are buffered, and only stored at commit time, forcing $O(|W|)$ commit complexity.

Because privileged transactions must never abort, whenever a conflict occurs between privileged and speculative transactions, the privileged transaction must always be favored. As mentioned in Section 2.2 we can either (a) suspend the commit of all speculative transactions while a privileged transaction is executing, sacrificing concurrency but allowing invisible reads; or b) use visible reads on privileged transactions, allowing the concurrent execution of privileged and speculative transactions. We chose b) so that concurrency (and therefore scalability) is not sacrificed.

Our system will therefore feature privileged transactions with visible reads and eager updates. Visible reads prevent concurrent speculative transactions from overwriting objects observed by the privileged transaction. However, this
is not the only conflict to be avoided. Because eager updates are used, privileged writes store the value directly to memory, before the transaction has committed. Speculative transactions can’t observe values written by live privileged transactions, because that would break atomicity (a transaction’s effects can only be observed in their entirety, or not observed at all). The existing algorithm [8, 9] prevents this case using write-locks: when a privileged transaction stores data in a location, it also acquires a lock to that location, which is released at commit time. Speculative transactions are unable to read from locations with such locks. Since all read and write-locks have to be released at commit time, the algorithm has \( O(|R| + |W|) \) commit complexity.

In order to obtain \( O(1) \) commit complexity we devised a novel algorithm, described in the next sections.

3.2 General Algorithm

Read and write-locks are used to ensure that the following conflicting cases don’t happen:

1. A transaction cannot overwrite an object that has been read by a live irrevocable transaction.
2. A transaction cannot observe writes made by a live privileged transaction.

The main idea in our algorithm is to use logical time to prevent these cases, instead of read and write-locks which force \( O(|R| + |W|) \) commit time. This is convenient because modern STM algorithms (including TL2, which we based ourselves in) already use logical time (in the form of a global-clock) to perform validation.

Each versioned-lock is augmented with another version number to support visible reads, so now they are composed of a mutual-exclusion lock with two version numbers: the write-\( ts \), which is the timestamp of the last transaction that wrote, and read-\( ts \), which is the timestamp of the last privileged transaction that read. We also divide the global clock’s domain in two: the even timestamps and the odd timestamps, and two distinct variables keep track of each: commit-\( ts \) for the even timestamps, and privileged-\( ts \) for the odd timestamps. Initially these are 2 and 1, respectively.

With these changes, the basic idea is that privileged transactions will always use odd timestamps, while the speculative transactions will always use even timestamps. This way, speculative transactions can check if they conflict with the privileged transaction, and the privileged transaction can commit by simply advancing the odd part of the global clock, immediately allowing speculative transactions to overwrite objects the privileged transaction read/wrote.

The privileged transaction’s algorithm is as follows:

1. **While the privileged transaction is running:**
   - **When a load (read) occurs**, the read-\( ts \) associated with the address to be loaded is set to privileged-\( ts \). If the associated versioned-lock is
acquired, we wait until it is unlocked. It might be acquired by a con-
current speculative transaction that is committing, so we wait until the
lock is released to observe all of the speculative transaction’s writes if it
commits, or none if it ends up aborting.

- **When a store (write) occurs**, we atomically set the associated \( write-ts \)
  value to \( commit-ts + 1 \) (an odd timestamp). This prevents speculative
  transactions from observing/overwriting our write until we commit.

2. **When the privileged transaction commits:**
   We atomically increment \( commit-ts \) by 2 and advance \( privileged-ts \) to the
   highest odd timestamp that is less than the new \( commit-ts \)’ (i.e. pre-increment
   \( commit-ts + 1 \)). By advancing \( privileged-ts \), speculative transactions can now
   overwrite the privileged transaction’s reads/writes and observe its writes.

The speculative transaction’s algorithm is very similar to the original, with
some added checks to detect conflicts with live privileged transactions:

1. **While the speculative transaction is running:**
   - **When a load (read) occurs**, the versioned-lock associated with the
     address to be loaded is checked. The original check is to abort the trans-
     action if \( write-ts \) is greater than \( tx-version \). However now we might have
     to abort even if \( write-ts \) is not greater than \( tx-version \) due to a privi-
     leged transaction, so we perform an additional check: abort if \( write-ts \)
     is greater than \( privileged-ts \) and \( write-ts \) is odd, i.e. there is a live privi-
     leged transaction that wrote, so we can not observe the value until the
     privileged transaction commits.

2. **When the speculative transaction commits:**
   Originally, the first step is to acquire all the locks in the write-set, abort-
   ing if we can not acquire all. Again, because now there can be a privileged
   transaction executing we must perform an additional check: we also abort
   if despite having acquired the lock, \( read-ts \) is equal to \( privileged-ts \) (mea-
   ning there is a live privileged transaction that read), or \( write-ts \) is greater
   than \( privileged-ts \) and \( write-ts \) is odd (meaning there is a live privileged
   transaction that wrote).

4 Implementation Details

We based our implementation in the TinySTM system and added modifications
to support privileged transactions. Every transaction has a \( privileged \) attribute,
which starts out being false. In order for a transaction to become privileged,
it must acquire (using CAS) a global lock. This prevents multiple privileged
transactions running concurrently: at any time, there can only be one (or none)
privileged transaction running.

We altered the versioned lock data structure: in tinySTM each memory loca-
tion is associated with a versioned-lock, which has a lock bit and a write-version
(named \( write-ts \) in the pseudo-code algorithms), indicating the last time the
location’s value was modified. Because privileged transactions use visible reads in our solution, we need to add one more value to the versioned lock: a read-version (named \textit{read-ts} in the algorithms), which is updated when a privileged transaction reads the location’s value.

Originally, the versioned lock occupied one memory word, divided into lock bit and write version. We can introduce a read version in one of two ways: using a separate word for it, or fitting it inside the original word, dividing it one more time. If we only use one word, setting the read version on a privileged read has to be done with CAS: we must read the word, and then store in its place the new read version concatenated with the old write version and lock bit, making sure none of these changed in the process (hence using CAS). On the other hand, if \textit{read-ts} is in a separate word, a simple store to that word is enough: since there can only be a maximum of one privileged transaction executing, there will be no concurrent writes on the word. However, the value must be immediately seen by other threads, which means that a \textit{memory barrier}, which flushes the local cache, is needed after the store operation. We didn’t reach an absolute conclusion about which option should be taken.

We also had to decide on how to access the global variable \textit{privileged-ts}. Speculative transactions use this value to verify, on every read operation, if the location they are trying to observe has been modified by a live privileged transaction (in which case the speculative transaction aborts). The first approach would be to load the value from memory on every read operation. However, because \textit{privileged-ts} is a global value being concurrently stored and loaded by separate threads, \textit{acquire and release semantics} are needed for reliable memory access. Performing a \textit{load-acquire} on every read operation would add significant latency, but a wiser approach can be taken: when a speculative transaction starts it loads \textit{privileged-ts} into a thread-local variable, and uses that value on subsequent read operations. This is more efficient, but a loss of information is introduced: the local snapshot of \textit{privileged-ts} can be outdated from the actual value, causing commits from privileged transactions to be missed. Because of this, when a speculative transaction detects a conflict with a privileged one, it might not be a conflict anymore, as the privileged transaction had committed. The simple solution to this is re-validating the snapshot when a conflict is detected.

There is also a decision to be made regarding the policy that decides which transactions become privileged, and when. As previously said, progress is guaranteed by turning systematically aborted transactions into privileged ones. There are many possible policies, which can take into account the number of times a transaction has aborted, which transaction has started the earliest, and other factors. We implemented a basic policy: every time a transaction aborts, it tries to become privileged by acquiring the global privileged-lock using CAS. If it does so successfully, the transaction runs in privileged mode and is guaranteed to commit. If it fails, it runs in speculative mode again.
5 Evaluation

In order to test our algorithm, we also implemented the previously existing algorithm, in which irrevocable transactions acquire read and write-locks on read and write operations. This is better than using the existing implementation because we can minimise differences between the two algorithms, so that experimental results vary only due to the algorithmic differences.

All tests were done on a server with four AMD Opteron 6272 CPU, each having sixteen logical cores (in total 64 concurrent threads are possible). The server is equipped with 64 GB RAM.

5.1 Linked List benchmark

To compare our algorithm with the existing one, we start by using the linked list microbenchmark. A linked list is initialized with a given size, and three types of transactions are run: insertions, which add an element to the list; deletions, which remove an element; and searches, which search the list for a specific element. Insertions and deletions are considered update or write transactions.

The list must be traversed one element at a time. If, for example, a transaction removes the 50th element from the list, it would have read 50 locations. The average number of reads a transaction makes is larger if the list’s initial size is larger.

We use this benchmark to compare the throughput of irrevocable transactions. We run it using only one thread, in privileged (irrevocable) mode. As we increase the list’s initial size and therefore the number of reads, we expect that the read and write-locks algorithm — which must acquire read-locks and release them one by one at commit time — performs worse than our "privileged" algorithm, which has a one-step commit.

Figure 1a shows that for a low initial list size (256) the existing read and write-locks algorithm performs the best. This is likely due to overhead from our algorithm. However, when the initial size is larger (32768 as shown on figure 1b) our algorithm outperforms the existing one. A bigger size means more values are read in an average transaction, and the cost of releasing the read-locks becomes noticeable.

5.2 STMBench7 benchmark

To further evaluate the algorithm, we used the STMBench7 benchmark, specified in [6]. This benchmark starts by creating a complex graph, and transactions perform operations concurrently on that graph. These operations can be long traversals, short traversals, or structural modification to the graph. The workload type is modifiable, and can be read-mostly, read-write, or write-mostly.

Figures 2a and 2b show that independently of the workload type, our algorithm performs similarly to the existing one, and both perform (predictably) much better than the one with no irrevocability, in which all threads run speculatively.
Fig. 1: Throughput of linked-list

(a) Initial size of 256
(b) Initial size of 32768

Fig. 2: Throughput of STMBench7

(a) Read-mostly workload
(b) Write-mostly workload
6 Summary and Future Work

We conceived a novel algorithm which is the basis for an STM implementation that can provide local progress and opacity simultaneously, and more efficiently than existing approaches. The key idea was to use logical time instead of locks for irrevocable transactions, improving their commit complexity.

Unfortunately, our experimental results don’t reveal a definitive advantage to our system. However, there are still improvements to be made. It would be interesting to establish smart policies about which transactions become privileged, and when. It’s likely that the best policy is different depending on the workload (high or low contention, read or write intensive), and the system could dynamically adapt based on the load it observes.

Additionally, more benchmarks and tests could be used to evaluate the system, specifically ones that feature large transactions, with big read and write sets. We experienced great variation due to the non-determinism of benchmarks, especially when using the big data-structure setting of STMBench7.

This work was a useful Undergraduate Research Opportunity, as it enlightened me not only on the techniques used for synchronization and concurrency management, but also on current hardware architecture and design, which must be taken into account when developing these systems.

Acknowledgments

The work and results reported in this paper result from a collaborative effort with Tiago Vale (t.vale@campus.fct.unl.pt) and João Lourenço (joao.lourenco@fct.unl.pt), who advised and supervised me during both the conceptual and experimental phases of my work.

References