STM systems: Enforcing strong isolation between transactions and non-transactional code

Tyler Crain[1], Eleni Kanellou[1], and Michel Raynal[1,2]

[1] IRISA, Université de Rennes 35042 Rennes Cedex, France
[2] Institut Universitaire de France
{tyler.crain,eleni.kanellou,michel.raynal}@irisa.fr

Abstract. Transactional memory (TM) systems implement the concept of an atomic execution unit called *transaction* in order to discharge programmers from explicit synchronization management. But when shared data is atomically accessed by both transaction and non-transactional code, a TM system must provide *strong isolation* in order to overcome consistency problems. Strong isolation enforces ordering between transactions and non-transactional operations and preserves transaction atomicity even with respect to non-transactional code. This paper presents a TM algorithm that implements strong isolation with the following features: (a) concurrency control of non-transactional operations is not based on locks and is particularly efficient, and (b) any non-transactional read or write operation always terminates (there no notion of commit/abort associated with them).

Keywords: Consistency, Non-transactional Operation, Strong Isolation, Terminating Operation, Transaction Atomicity, Transactional Memory, Opacity, Strong Isolation, TL2.

1 Introduction

STM Systems. Transactional Memory (TM) [8] [15] has emerged as an attempt to allow concurrent programming based on sequential reasoning: By using TM, a user should be able to write a correct concurrent application, provided she can create a correct sequential program. The underlying TM system takes care of the correct implementation of concurrency. However, while most existing TM algorithms consider applications where shared memory will be accessed solely by code enclosed in a transaction, it still seems imperative to examine the possibility that transactional and non-transactional code could co-exist.

Strong vs Weak Isolation. TM has to guarantee that transactions will be isolated from each other, but when it comes to transactions and non-transactional operations, there are two paths a TM system can follow: it may either act oblivious to the concurrency between transactions and non-transactional operations, or it may take this concurrency into account and attempt to provide isolation guarantees even between transactional and non-transactional operations. The

first case is referred to as *weak isolation* while the second case is referred to as *strong isolation*. (This distinction of guarantees was originally made in [11], where reference was made to "weak atomicity" versus "strong atomicity".)

Under weak isolation, a TM algorithm would be used without much overhead alongside code that contains non-transactional operations. Then, a non-transactional read operation on shared variable x would be able to observe a write operation on x which is performed by a transaction that has not yet committed. Furthermore, a read operation on x performed by a live transaction would be able to observe updates on variable x that happen by non-transactional code during this transaction's execution. While this behavior clearly violates the isolation principle of the transaction abstraction, it could nevertheless be anticipated and used appropriately by the programmer, still resulting in correctly functioning applications. This would require the programmer to be conscious of eventual race conditions between transactional and non-transactional code.

Desirable Properties. In order to keep consistent with the spirit of TM principles, however, a system should prevent unexpected results from occurring in presence of race conditions. Furthermore, concurrency control should ideally be implicit and never be delegated to the programmer [2,12]. These are the reasons for which strong isolation is desirable. Under strong isolation, the aforementioned scenarios, where non-transactional operations violate transaction isolation, would not be allowed to happen. An intuitive approach to achieving strong isolation is to treat each non-transactional operation that accesses shared data as a "mini-transaction", i.e., one that contains a single operation. In that case, transactions will have to be consistent (see Sect. 2) not only with respect to each other, but also with respect to the non-transactional operations. However, while the concept of the memory transaction includes the possibility of abort, the concept of the non-transactional operation does not. This means that a programmer expects that a transaction might fail, either by blocking or by aborting. Non-transactional accesses to shared data, though, will usually be read or write operations, which the programmer expects to be atomic. While executing, a read or write operation is not expected to be de-scheduled, blocked or aborted.

Content of the Paper. This paper presents a TM algorithm which takes the previous issues into account. It is built on top of TM algorithm TL2 [5], a word-based TM algorithm that uses locks. More precisely, TL2 is modified to accept non-transactional read and write operations and strong isolation in their presence. However, the algorithm is designed without the use of locks for non-transactional code, in order to guarantee that their execution will always terminate. To achieve this, two additional functions are specified, which substitute conventional read or write operations that have to be performed outside of a transaction. Possible violations of correctness under strong isolation are reviewed in Sect. 2. The TL2 algorithm is described in Sect. 3. Section 4 describes the proposed algorithm that implements strong isolation for TL2, while Sect. 5 concludes the paper by summarizing the work and examining possible applications.

2 Correctness and Strong Isolation

Consistency Issues. When it comes to environments where shared memory is accessed exclusively through transactions, then most accepted consistency conditions build on the idea of *serializability* [13], a condition first established for the study of database transactions. For a concurrent execution of transactions to be serializable, it must be possible to find a serialization for it, i.e., a legal sequential execution that is equivalent to it. Serializability refers to committed transactions, however. In the context of memory transactions, stricter criteria are desirable, because even transactions that will eventually abort may cause program exceptions if they observe an inconsistent state of the shared memory. For this reason, a prominent consistency condition for transactional memory, which is stricter than serializability, has beeb proposed. This consistency condition, which is called *opacity* [6], requires that both committed as well as aborted transactions observe a consistent state of memory. This implies that in order for a concurrent execution of memory transactions to be opaque, there must exist an equivalent, legal sequential execution that includes both committed transactions and aborted transactions, albeit reduced to their read prefix. Other consistency conditions have been proposed. Among them, *virtual world consistency* [9] is weaker than opacity while keeping its spirit (i.e., it depends on both committed transactions and aborted transactions).

Transaction vs Non-transactional Code. In a concurrent environment, shared memory may occasionally be accessed by both transactions as well as non-transactional operations. This is mostly brought about by the need to continue using legacy code that existed before transactional memory was implemented. Traditionally, however, transactions are designed to synchronize with other transactions and do not take the possibility of non-transactional code into account. Traditionally, the same goes for non-transactional code: It is not expected from the programmer to know the synchronization details of the transactional memory algorithm that will run concurrently with her non-transactional code. Therefore, this possibility of *co-existence of two different paradigms*, as well as the fact that transactional memory is mostly implemented as a software platform - instead of the transaction abstraction being directly provided by the hardware - reveal two different aspects that transactional memory may acquire: In the first aspect, it is the only way through which shared memory may be accessed by concurrent processes. In the second aspect, which comes into view when it exists alongside non-transactional code, it is just another means of achieving synchronization, along with locks, fences and other traditional methods[3].

For that first aspect of transactional memory, consistency is guaranteed if the transactional memory system implements opacity. For the second aspect, however, even if a transactional memory system implements opacity, consistency violations may still be possible in the presence of concurrent non-transactional

[3] See also the debate about whether transactions should be implemented through the use of locks or whether locks should be implemented through the use of transactions.

code. It can be acceptable to have concurrent environments that may be prone to some types of violations, as is the case with systems that provide weak isolation [11] [16]. Under weak isolation, transactional and non-transactional operations can be concurrent.

However, as indicated in the Introduction, transactions are considered to happen atomically only with respect to other transactions. It is possible for non-transactional operations to see intermediate results of transactions that are still live. Conversely, a transaction may see the results of non-transactional operations that happened during the transaction's execution. If this behavior is not considered acceptable for an application, then the responsibility to prevent it is delegated to the programmer of concurrent applications for this system. However, in order to spare the programmer this responsibility, both the transactional memory algorithm as well as the non-transactional read and write operations must be implemented in a way that takes their co-existence into account. Such an implementation that provides synchronization between transactional and non-transactional code is said to provide strong isolation.

Privatization/Publication. In a system that does not provide strong isolation, unsynchronized concurrent access can occur between two processes on a memory area that both view as shared. However, it can also occur on memory areas that one of the threads views as shared while the other considers them to be its private memory area. This is referred to as the *privatization problem*. An area of shared memory is privatized, when a process that modified it makes it inaccessible to other concurrent processes[4] [18]. A typical example of privatization would be the manipulation of a shared linked list. The removal of a node by a transaction (for private use by the process that invoked the transaction) through non-transactional code, constitutes privatization. A transaction that privatizes a memory area is called privatizing transaction.

Consistency problems arise if for example, the process that privatized a memory area starts accessing it non-transactionally, while the privatization has not become visible to other processes, which access that area through transactions, considering that it is still shared. As shown in [18], incorrect results due to privatization can occur regardless of the update policy that a transactional memory algorithm implements.

An STM implementation that is designed to prevent memory inconsistencies that may arise during privatization is called *privatization safe*. Several solutions have been proposed for the privatization problem such as using visible reads, conventional means of synchronization such as fences, sandboxing, partitioning by consensus [14], the use of lock-free reference counters [1] or by using private transactions [4], to name a few.

A system that provides *strong isolation* has the advantage of inherently also solving the privatization problem. Strong isolation always imposes synchroniza-

[4] Conversely, a memory area is made public when it goes from being exclusively accessible by one process to being accessible by several processes [17] . This is referred to as the *publication problem* and the consistency issues that arise are analogous.

tion between transactional and non-transactional code that access memory areas that are potentially shared, which means that a system that guarantees strong isolation is inherently privatization safe.

Providing Strong Isolation. There are different definitions in literature for strong isolation [11] [10] [7]. In the present paper we will consider that strong isolation as follows:

1. Non-transactional operations are considered as "mini" transactions which never abort and contain only a single read or write operation.
2. The consistency condition for transactions is opacity.

As non-transactional read an write operations never abort, this is called *terminating strong isolation* in the following.

The first item implies what is referred to as *containment* and *non-interference* [11]. Containment is illustrated in the left part of Fig. 1. There, under strong isolation, we have to assume that transaction T_1 happens atomically, i.e.,"all or nothing", also with respect to non-transactional operations. Then, while T_1 is alive, no non-transactional read, such as R_x, should be able to obtain the value written to x by T_1. Non-interference is illustrated in the right part of Fig. 1. Under strong isolation, non-transactional code should not interfere with operations that happen inside a transaction. Therefore, transaction T_1 should not be able to observe the effects of operations W_x and W_y, given that they happen concurrently with it.

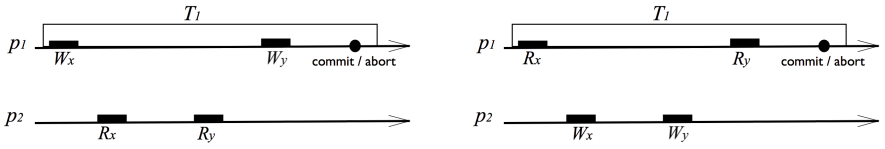

Fig. 1. Left: *Containment* (operation R_x should not return the value written to x inside the transaction). Right: *Non-Interference* (wile it is still executing, transaction T_1 should not have access to the values that were written to x and y by process p_2).

Non-interference violations can be caused by the ABA problem. Consider the case where a transaction T pertaining to process p_1 reads variable x through read operation R_x, and finds that it contains value v_1. Before T commits, and after R_x has completed, assume that another process p_2 modifies x by writing value v_2 to it through non-transactional write operation W_{1x}. Then, assume that either the same process p_2 or a different process p_3 write non-transactionally value v_1 to x through operation W_{2x}. In this case, process $p1$ should have a means to detect this occurrence and transaction T should not commit, given that otherwise, strong isolation would be violated.

Non-interference for a transaction T_1 can also be compromised by interaction between non-transactional operations and another transaction T_2, as illustrated in Fig. 2. There, non-transactional operation R_{2x} reads what transaction T_2 has written to shared variable x. Due to maintaining consistency, it is not possible to find a correct serialization order where non-transactional operation W_y does not happen during the duration of transaction T_1, violating non-interference. Therefore, in order to preserve opacity, this situation would have to be detected when transaction T_1 attempts to execute operation R_y and T_1 would have to be aborted.

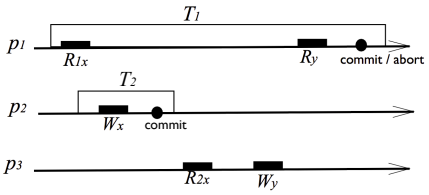

Fig. 2. Transaction T_2 and non-transactional operations of process p_3 interfere with transaction T_1.

3 A Brief Presentation of TL2

TL2, aspects of which are used in this paper, has been introduced by Dice, Shalev and Shavit in 2006 [5]. The word-based version of the algorithm is used, which means that the shared variables accessed are memory words and that the operations issued by a transaction are simply read and write operations. For the sake of brevity, implementation details of the algorithm are omitted in the following description.

Main Features of TL2. The shared variables that a transaction reads form its *read set*, while the variable it has to update, the *write set*. Read operations in TL2 are *invisible*, meaning that when a transaction reads a shared variable, there is no indication of that fact towards other transactions. Write operations are *deferred*, meaning that TL2 does not perform the updates as soon as it "encounters" the shared variables that it has to write to. Instead, the updates it has to perform are logged into a local list (also called *redo log*) and are applied to the shared memory only once the transaction is certain to commit. To control transaction synchronization, TL2 employs locks and logical dates.

Locks and Logical Time. A lock is associated with each shared variable. A transaction first has to obtain the locks of the variables of its write set, before it can update them. This is attempted when the transaction attempts to commit. Furthermore, a transaction has to check the logical dates of the variables in its read

set, in order to ensure that the values it has read correspond to a consistent snapshot of shared memory. TL2 implements logical time as an integer counter denoted GVC. When it starts up, a transaction reads the current value of GVC into local variable, rv. When a transaction attempts to commit, it performs an increment-and-fetch on GVC, and stores the return value in local variable wv (which can be seen as a write version number or a version timestamp). Should the transaction commit, it will assign its wv as the new logical date of the shared variables in its write set. A transaction must abort if its read set is not valid. When this occurs, then the logical date of every item on the read set is less than the transaction's rv value. If, on the contrary, the logical date of a read set item is larger than the rv of the transaction, then this indicates that, in the meanwhile, a concurrent transaction has updated it and committed.

4 Implementing Terminating Strong Isolation

A possible solution to the problem of ensuring isolation in the presence of non-transactional code consists in using locks: Each shared variable would then be associated with a lock and both transactions as well as non-transactional operations would have to acquire the lock before accessing the variable. In order to achieve strong isolation, transactions would have to acquire the locks of the variables both in their read and in their write set.

Locks are already used in TM algorithms - such as TL2 itself - where it is however assumed that shared memory is only accessed through transactions. The use of locks in a TM algorithm entails blocking and may even lead a process to starvation. However, it can be argued that these characteristics are acceptable, given that the programmer accepts the fact that a transaction has a duration and that it may even fail: The fact that there is always a possibility that a transaction will abort means that the eventuality of failure to complete can be considered a part of the transaction concept.

On the contrary, when it comes to read or write accesses to a shared variable, a non-transactional operation is understood as an event that happens atomically and completes. Unfortunately strong isolation implemented with locks entails the blocking of non-transactional read and write operations.

Given that this approach would be rather counter-intuitive for the programmer (as well as possibly detrimental for program efficiency), the algorithm presented in this section provides a solution for adding strong isolation which does not based on locks for the execution of non-transactional operations. This algorithm builds on the base of TM algorithm TL2 and extends it in order to account for non-transactional operations. While read and write operations that appear inside a transaction follow the original TL2 algorithm rather closely (commit-time lock acquisition, write-back and validation of the read set), the proposed algorithm specifies specific non-transactional read and write operations that are to be used by the programmer, substituting conventional shared memory read and write operations.

4.1 Memory Setup and Data Structures

Memory Setup. The underlying memory system is made up of atomic read/write registers. Moroever some of them can also be accessed by the the following two operations. The operation denoted Fetch&increment() atomically adds one to the register and returns its previous value. The operation denoted C&S() (for compare and swap) is a conditional write. C&S(x, a, b) writes b into x iff $x = a$. In that case it returns *true*. Otherwise it returns *false*.

The proposed algorithm assumes that the variables are of types and values that can be stored in a memory word. This assumption aids in the clarity of the algorithm description but it is also justified by the fact that the algorithm extends TL2, an algorithm that is designed to be word-based.

As in TL2, the variable GVC acts as global clock which is incremented by update transactions. Apart from a global notion of "time", there exists also a local one, and therefore, each process maintains a local variable denoted *time*, which is used in order to keep track of when, with respect to the GVC, a non-transactional operation or a transaction was last performed by the process. This variable is then used during non-transactional operations to ensure the linearization of operations is not violated. Each process's *time* variable is therefore incremented both during transactional and non-transactional operations.

In TL2 a shared array of locks is maintained and each shared memory word is associated with a lock in this array usually by some hash fuction. Given this, a memory word directly contains the value of the variable that is stored in it. Instead, the algorithm presented here, uses a different memory setup is used that does not require a lock array, but does requires an extra level of indirection when loading and storing values in memory. Instead of storing the value of a variable directly to a memory word, each write operation on variable *var*, transactional or non-transactional, creates an algorithm-specific structure that contains the new value of *var*, as well as necessary meta-data.

T-record and NT-record. The algorithm-specific data structures are shared and can be of either two kinds, which will be referred to as T-records and NT-records. A T-record is created by a transactional write operation while an NT-record is created by a non-transactional write operation. A memory word that is used to store variable *var* at address *addr* will then contain a pointer to a record of either of the aforementioned types and within this structure the actual value for *var* is stored as a field. This is illustrated in Figure 3.

New T-records are created during the transactional write operations. Then during the commit operation the pointer stored at *addr* is updated to point to this new T-record. During NT-write operations new NT-records are created and the pointer at *addr* is updated to point to the records.

When a read operation - be it transactional or non-transactional - accesses a shared variable it cannot know beforehand what type of record it will find on the list. Therefore, it can be seen in the algorithm listings, that whenever a record from the list is accessed, the operation checks its type, i.e., it checks whether it

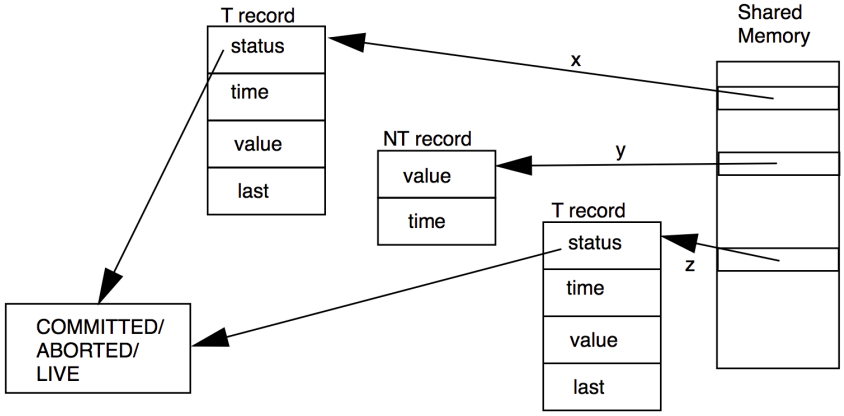

Fig. 3. The memory setup and the data structures that are used by the algorithm.

is a T-record or an NT-record (for example, line 02 in Figure 4 contains such a check. A T-record is "of type T", while an NT-record is "of type NT").

T-record. A T-record is a structure containing the following fields.

status This field indicates the state of the transaction that created the T-record. The state can either be LIVE, COMMITTED or ABORTED. The state is initially set to LIVE and is not set to COMMITTED until during the commit operation when all locations of the transaction's write set have been set to point to the transaction's T-records and the transaction has validated its read set.
Since a transaction can write to multiple locations, the *status* field does not directly store the state, instead it contains a pointer to a memory location containing the state specific to a single transaction. Therefore the *status* field of each T-record created by the same transaction will point to the same location. This ensures that any change to the transaction's state is immediately recognized at each record.

time The *time* field of a T-record contains the value of the GVC at the moment the record was inserted into the list of records. This is similar to the logical time values of TL2 that are stored in the lock array.

value This field contains the value that is meant to be written to the chosen memory location.

last During the commit operation, locations are updated to point to the committing transaction's T-records, overwriting the previous value that was stored in this location. Validation or a concurrent non-transactional operations may cause this transaction to abort after it update some memory locations, but before it fully commits. Due to this, the previous value of the location needs to be available for future reads. Instead of rolling back old memory values, the *last* field of a T-record is used, storing the previous value of this location.

NT-record. An NT-record is a structure containing the following fields.

value This field contains the value that is meant to be written to the chosen memory location.

time As in the case of T-records, the *time* field of NT-records also stores the value of the *GVC* when the write took place. This field is used to avoid inconsistencies such as the ones illustrated by Figure 2.

Due to this different memory structure a shared lock array is no longer needed, instead of locking each location in the write set during the commit operation, this algorithm performs a compare and swap directly on each memory location changing the address to point to one of its T-records. After a successful compare and swap and before the transactions status has been set to COMMITTED or ABORTED, the transaction effectively owns the lock on this location. Like in TL2, any concurrent transaction that reads the location and sees that it is locked ($status$ = LIVE) will abort itself.

During read operations (transactional or non-transactional) the additional data in these structures is used to determine the most recent valid value of a variable. Should the item be of type NT-record, then it's *value* field contains the most recent valid value of the variable. On the other hand, should the item be a T-record, then if the *status* field of the record is equal to COMMITTED, the *value* field represents the current value of the variable. Otherwise, the *last* field contains the current value.

Transactional Read and Write Sets. Like TL2, read only transactions do not use read sets while update transactions do. The read set is made up of a set of tuples for each location read, $\langle addr, value \rangle$ where *addr* is the address of the location read and *value* is the value read from the location. The write set is also made up of tuples for each location written by the transaction, $\langle addr, item \rangle$ where *addr* is the location to be written and *item* is a T-record that will be pointed to by *addr* during the commit operation.

Discussion. One advantage of the TL2 algorithm is in its memory layout. This is because reads and writes happen directly to memory (without indirection) and the main amount of additional memory that is used is in the lock array. Unfortunately this algorithm breaks that and requires an additional level of indirection as well as additional memory per location. These additional requirements can be an acceptable trade-off given that they are only needed for memory that will shared between transactions. Still, in the technical report [3] we present two variations of the algorithm that trade off different memory schemes for different additional costs to the transactional and non-transactional operations.

4.2 Description of the Algorithm

The main goal of the algorithm is to provide strong isolation in such a way that the non-transactional operations are never blocked. In order to achieve this,

the algorithm delegates most of concurrency control and consistency checks to the transactional code. Non-transactional operations access and modify memory locations without waiting for concurrent transactions and it is mainly up to transactions accessing the same location to do it in a way that ensures safe concurrency - and to abort if this is not possible. As a result, this algorithm gives high priority to non-transactional code.

Given the particular memory arrangement that the algorithm uses, pointers are used in order to load and store items from memory. [5]

4.3 Non-transactional Operations

In order to comply with the algorithm a programmer has to use the algorithm-specific read and write operations when a shared variable has to be accessed outside of a transaction. This be done by hand or applied by a complier. Algorithms for these operations is presented in Figure 4.

```
operation non_transactional_read(addr) is
(01)   tmp ← load(addr);
(02)   if ( tmp is of type T ∧tmp.status ≠ COMMITTED )
(03)     then if (tmp.time ≤ time ∧ tmp.status = LIVE)
(04)            then C&S(tmp.status, LIVE, ABORTED) end if;
(05)          if (tmp.status ≠ COMMITTED)
(06)            then value ← tmp.last
(07)            else value ← tmp.value
(08)          end if;
(09)     else value ← tmp.value
(10)   end if;
(11)   time ← max(time, tmp.time)
(12)   if (time = ∞) then time = GCV end if;
(13)   return (value)
end operation.

operation non_transactional_write(addr, value) is
(14)   allocate new variable next_write of type NT;
(15)   next_write ← (addr, value, ∞);
(16)   store(addr, next_write)
(17)   time ← GVC;
(18)   next_write.time ← time;
end operation.
```

Fig. 4. Non-transactional operations for reading and writing a variable.

Non-transactional Read. The operation non_transactional_read() is used to read, when not in a transaction, the value stored at *addr*. The operation first dereferences the pointer stored at *addr* (line 01). If the item is a T-record that was created by a transaction which has not yet committed then the *value* field cannot be immediately be read as the transaction might still abort. If the current process has read a value that is more recent then the transaction (meaning the

[5] The following notation is used. If pt is a pointer, $pt\downarrow$ is the object pointed to by pt. if aa is an object, $\uparrow aa$ is a pointer to aa. Hence $((\uparrow aa)\downarrow = aa$ and $\uparrow (pt\downarrow) = pt$.

process's *time* field is greater or equal to the T-records *time*, line 03) then the transaction must be directed to abort (line 04) in order to insure linearizability (containment specifically) is not violated.

From a T-record with a transaction that is not committed, the value from the *last* field is stored to a local variable (line 06) and will be returned on operation completion. Otherwise the *value* field of the T or NT-record is used (line 07).

The process local variable *time* is advanced to the maximal value among its current value and the logical date of the T or NT-record whose value was read. Finally if *time* was set to ∞ on line 11, then it is updated to the GCV on line 12. The updated *time* value is used by non-transactional operations and is necessary in order to allow the detection of consistency violations such as the one illustrated by Figure 2. Once these book-keeping operations are finished, the local variable *value* is returned (line 13).

Non-transactional Write. The operation non_transactional_write() is used to write to a shared variable *var* by non-transactional code. The operation takes as input the address of the shared variable as well as the value that has to be written to it as arguments. This operation creates a new NT-record (line 14), fills in its fields (line 15) and changes the pointer stored in *addr* so that it references the new record it has created (line 16). Unlike update transactions, non-transactional writes do not increment the global clock variable GCV. Instead they just read GCV and set the NT-record's time value as well as the process local *time* to the value read (line 17 and 18). Since the GCV is not incremented, several NT-records might have the same *time* value as some transaction. When such a situation is recognized where a live transaction has the same time value as an NT-record the transaction is aborted (if during an NT-read operation, line 04) or read set validation is performed (if during a transactional read operation, line 23 of figure 5). This is done in order to prevent consistency violations such as the one shown in Figure 2.

4.4 Transactional Read and Write Operations

The transactional operations for performing reads and writes are presented in Figure 5.

Transactional Read. The operation transactional_read() takes *addr* as an input argument. It starts by checking whether the desired variable already exists in the transaction's write set, in which case the value stored there will be returned (line 19). If the variable is not contained in the write set, the pointer in *addr* is dereferenced (line 20) and set to *tmp*. Once this is detected to be a T or NT-record some checks are then performed in order to ensure correctness.

In the case that *tmp* is a T-record the operation must check to see if the status of the transaction for this record is still LIVE and if it is the current transaction is aborted (line 29). This is similar to a transaction in TL2 aborting itself when a locked location is found. Next the T-record's *time* field is checked, and (similar to TL2) if it greater then the process's local *rv* value the transaction

must abort (line 32) in order to prevent consistency violations. If this succeeds without aborting then the local variable *value* is set depending on the stats of the transaction that created the T-record (line 29-30).

In case *tmp* is an NT-record (line 21), the operation checks whether the value of the *time* field is greater or equal to the process local *rv* value. If it is, then this write has possibly occurred after the start of this transaction and there are several possibilities. In the case of a read only transaction, the transaction is aborted and restarted as an update transaction (line 24). In the case of an update transaction validation must be preformed, ensuring that none of the values it has read have been updated (line 23). Finally local variable *value* is set to be the value of the *value* field of the *tmp* (line 26).

It should be noted that the reason why the checks are performed differently for NT-records and T-records is because the NT-write operations do not update the global clock value while update transaction do. This means that the checks must be more conservative in order to ensure correctness. If performing per value validation or restarting the transaction as an update transaction is found to be too expensive, a third possibility would be to just increment the global clock, then restart the transaction as normal.

Finally to finish the read operation, the $\langle addr, value \rangle$ is added to the read set if the transaction is an update transaction (line 34), and the value of the local variable *value* is returned.

```
operation transactional_read(addr) is
(19)    if addr ∈ ws then return (item.value from addr in ws) end if;
(20)    tmp ← load(addr);
(21)    if (tmp is of type NT)
(22)       then if (tmp.time >= rv)
        % Do validation to prevent abort due to a non-transactional write
(23)            then if this is an update transaction then validate_by_value()
(24)                 else abort() and restart as an update transaction end if;
(25)            end if;
(26)            value ← tmp.value;
(27)       else
(28)            if ((status ← tmp.status) ≠ COMMITTED )
(29)              then if (status = LIVE) then abort() else value ← tmp.last end if;
(30)              else value ← tmp.value
(31)            end if;
(32)            if (tmp.time > rv) then abort() end if;
(33)    end if;
(34)    if this is an update transaction then add ⟨addr, value⟩ to rs end if;
(35)    return (value)
end operation.

operation transactional_write(addr, value) is
(36)    if addr ∉ ws
(37)       then allocate a new variable item of type T;
(38)            item ← (value, status, ∞); ws ← ws ∪ ⟨addr, item⟩;
(39)       else set item.value with addr in ws to value
(40)    end if;
end operation.
```

Fig. 5. Transactional operations for reading and writing a variable.

Transactional Write. The transactional_write() operation takes *addr* as input value, as well as the value to be written to *var*. As TL2, the algorithm performs commit-time updates of the variables it writes to. For this reason, the transactional write operation simply creates a T-record and fills in some of its fields (lines 37 - 38) and also adds it to the transaction's write set. However, in the case that a T-record corresponding to *addr* was already present in the write set, the *value* field of the corresponding T-record is simply updated (line 39).

Begin and End of a Transaction Book-ending a transaction are operations begin_transaction() and try_to_commit(), which are presented in Figure 6. begin_transaction() initializes local variables that will be necessary for the execution of the transaction. This includes rv which is set to GCV and, like in TL2, is used during transactional reads to ensure correctness, as well as *status* which is set to LIVE and the read and write sets which are initialized as empty sets. (lines 41-43).

```
operation begin_transaction() is
(41)   determine whether transaction is update transaction based on compiler/user input
(42)   rv ← GVC; Allocate new variable status;
(43)   status ←LIVE;  ws ← ∅; rs ← ∅
end operation.

operation try_to_commit() is
(44)   if (ws = ∅) then return (COMMITTED) end if;
(45)   for each (⟨addr, item⟩ ∈ ws) do
(46)     tmp ← load(addr);
(47)     if (tmp is of type T ∧ (status ← tmp.status) ≠ COMMITTED )
(48)           then if (status = LIVE) then abort() else item.last ← tmp.last end if;
(49)           else item.last ← tmp.value
(50)     end if;
(51)     item.time ← tmp.time;
(52)     if (¬C&S(addr, tmp, item)) then abort() end if;
(53)   end for;
(54)   time ← fetch&increment(GVC); validate_by_value();
(55)   for each (⟨addr, item⟩ ∈ ws) do
(56)     item.time ← time;
(57)     if (item ≠ load(addr)) then abort() end if;
(58)   end for;
(59)   if C&S(status, LIVE, COMMITTED)
(60)     then return (COMMITTED)
(61)     else abort()
(62)   end if;
end operation.
```

Fig. 6. Transaction begin/commit.

After performing all required read and write operations, a transaction tries to commit, using the operation try_to_commit(). Similar to TL2, a try_to_commit() operation starts by trivially committing if the transaction was a read-only one (line 44) while an update transaction must announce to concurrent operations what locations it will be updating (the items in the write set). However, the algorithm differs here from TL2, given that it is faced with concurrently happening non-transactional operations that do not rely on locks and never block. This, in turn, implies that even after acquiring the locks for all items in its write set, a

transaction could be "outrun" by a non-transactional operation that writes to one of those items causing the transaction to be required to abort in order to ensure correctness. Therefore while TL2 locks items in its write set using a lock array, this algorithm compare and swaps pointers directly to the T-records in its write set (lines 45-53) while keeping a reference to the previous value. The previous value is stored in the T-record before the compare and swap is performed (lines 48-49) with a failed compare and swap resulting in the abort of the transaction. If while performing these compare and swaps the transaction notices that another LIVE transaction is updating this memory, it aborts itself (line 48).

The operation then advances the GVC, taking the new value of the clock as the logical time for this transaction (line 54). Following this the read set of the transaction is validated for correctness (line 54). Once validation has been performed the operation must ensure that non of its writes have been concurrently overwritten by non-transactional operations (lines 55-58) if so then the transaction must abort in order to (line 57) to ensure consistency. During this check the transaction updates the *time* value of its T-records to the transactions logical time (line 56) similar to the way TL2 stores counter values in the lock array so that future operations will know the linearization of this transaction's updates.

Finally the transaction can mark its updates as valid by changing its *status* variable from LIVE to COMMITTED (line 59). This is done using a compare and swap as there could be a concurrent non-transactional operations trying to abort the transaction. If this succeeds then the transaction has successfully committed, otherwise it must abort and restart.

```
operation validate_by_value() is
(63)    rv ← GVC;
(64)    for each ⟨addr, value⟩ in rs do
(65)      tmp ← load(addr);
(66)      if (tmp is of type T ∧tmp.status ≠ COMMITTED)
(67)            then if (tmp.status = LIVE ∧item ∉ ws) then abort() end if;
(68)              new_value ← tmp.last;
(69)            else new_value ← tmp.value
(70)      end if;
(71)      if new_value ≠ value then abort() end if;
(72)    end for;
end operation.

operation abort() is
(73)    status ← ABORTED;
(74)    the transaction is aborted and restarted
end operation.
```

Fig. 7. Transactional helper operations.

Transactional Helping operations. Apart from the basic operations for starting, committing, reading and writing, a transaction makes use of helper operations

to perform aborts and validate the read set. Pseudo-code for this kind of helper operations is given in Figure 7.

Operation validate_by_value() is an operation that performs validation of the *rs* of a transaction. Validation fails if any location in *rs* is currently being updated by another transaction (line 67) or has had its changed since it was first read by the transaction (line 71) otherwise it succeeds. The transaction is immediately aborted if validation fails (lines 67, 71). Before the validation is performed the read version (*rv*) is updated to be the current value of the global clock (line 63). This is done because the transaction will be valid at this time if validation succeeds and a larger clock value could prevent future validations and aborts.

When a transaction is aborted in the present algorithm, the status of the current transaction is set to ABORTED (line 73) and it is immediately restarted as a new transaction.

5 Conclusion

This paper has presented an algorithm that achieves non-blocking strong isolation "on top of" a TM algorithm based on logical dates and locks, namely TL2. In case of conflict of a transactional and a non-transactional operation on a shared variable, this algorithm gives priority to the non-transactional operation, reasoning that while an eventual abort or restart is part of the specification of the transaction, this is not the case for conventional read or write operations. Due to this priority mechanism, the proposed algorithm is particularly appropriate for environments in which processes do not rely heavily on the use of especially large transactions. In such environments, strong isolation is provided for transactions, while conventional read and write operations execute with a small overhead.

References

1. Afek, Y., Avni, H., Dice, D., Shavit, N.: Efficient lock free privatization. In: *Proc. 14th Int'l conference on Principles of Distributed Systems (OPODIS'10)*, pp. 333–347, Springer-Verlag, LNCS #6490 (2010)
2. Crain, T., Imbs, D., Raynal, M.: Towards a Universal Construction for Transaction-based Multiprocess Programs. In: *Proc. 13th Int'l Conference on Distributed Computing and Networking (ICDCN'12)*, pp. 61–75, Springer Verlag LNCS #7129 (2012)
3. Crain, T., Kanellou, E., Raynal, M.: Enforcing Strong Isolation. Irisa Technical Report (2012)
4. Dice, D., Matveev, A., Shavit, N.: Implicit privatization using private transactions. In: *Proc. Workshop on transactional memory (TRANSACT'10)* (2010)
5. Dice, D., Shalev O., Shavit, N.: Transactional Locking II. In: *Proc. 20th Int'l Symposium on Distributed Computing (DISC'06)*, Springer-Verlag, LNCS #4167, pp. 194–208 (2006)
6. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: *Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, ACM Press, pp. 175–184 (2008)

7. Harris, T., Larus, J., Rajwar, R.: Transactional Memory, *2nd edition, Synthesis Lectures on Computer Architecture*, Morgan & Claypool Publishers (2006)
8. Herlihy, M., Moss J.M.B.: Transactional memory: architectural support for lock-free data structures, In: *Proc. of the 20th annual Int'l Symposium on Computer Architecture (ISCA '93)*, ACM Press, pp. 289–300 (1993)
9. Imbs, D., Raynal, M.: A versatile STM protocol with invisible read operations that satisfies the virtual world consistency condition. In: *16th Colloquium on Structural Information and Communication Complexity (SIROCCO'09)*, Springer Verlag LNCS #5869, pp. 266–280 (2009)
10. Maessen, J.-W., Arvind, M.: Store Atomicity for Transactional Memory. *Electronic Notes on Theoretical Computer Science*, 174(9):117–137 (2007).
11. Martin, M., Blundell, C., Lewis, E.: Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, 5(2): (2006)
12. Matveev, A., Shavit, N.: Towards a Fully Pessimistic STM Model. In: *Proc. Workshop on transactional memory (TRANSACT'12)* (2012)
13. Papadimitriou, Ch.H.: The Serializability of Concurrent Updates. *Journal of the ACM*, 26(4):631–653 (1979)
14. Scott, M.L., Spear, M.F., Dalessandro, L., Marathe, V.J.: Delaunay Triangulation with Transactions and Barriers. In: *Proc. 10th IEEE Int'l Symposium on Workload Characterization (IISWC '07)*, IEEE Computer Society, pp. 107–113 (2007)
15. Shavit, N., Touitou, D.: Software transactional memory. Software Transactional Memory. In: *Distributed Computing*, 10(2):99–116 (1997)
16. Shpeisman, T., Menon, V., Adl-Tabatabai, A.R., Balensiefer, S., Grossman, D., Hudson, R.L., Moore, K.F., Saha, B.: Enforcing isolation and ordering in STM. In: *ACM SIGPLAN Noticers*, 42(6):78–88 (2007)
17. Spear M.F., Dalessandro L., Marathe V.J., Scott, M.L.: Ordering-Based Semantics for Software Transactional Memory. In: *Proc 12th Int'l Conference on Principles of Distributed Systems (OPODIS '08)*, Springer-Verlag LNCS #5401, pp. 275–294 (2008)
18. Spear, M.F., Marathe, V.J., Dalessandro, L., Scott, M.L.: Privatization techniques for software transactional memory. In: *Proc. 26th annual ACM symposium on Principles of Distributed Computing (PODC '07)*, . ACM press, pp. 338–339, (2007)

Appendix

6 Version of algorithm that does not use NT-records

This algorithm also provides wait-free NT read and write operations. The difference is that NT-records are not used. Instead NT values are read and written directly from memory. By doing this, memory allocations are not needed in NT writes and NT reads have one less level of indirection.

The cost of this is more frequent validations required in transactions when conflicts with NT writes occur. This algorithm is shown in Figs. 8-10.

7 Version of algorithm with non-blocking NT-reads and blocking NT-writes

This algorithm allows wait-free NT read operations. The only change that is needed to the base TL2 algorithm is that when an item is locked it points to

```
operation non_transactional_read(addr) is
(75)    tmp ← load(addr);
(76)    if ( tmp is of type T )
(77)      then if (tmp.status = LIVE)
(78)             then C&S(tmp.status, LIVE, ABORTED)
(79)           end if;
(80)           if (tmp.status = ABORTED)
(81)             then value ← tmp.last
(82)             else value ← tmp.value
(83)           end if;
(84)      else value ← tmp
(85)    end if;
(86)    return (value)
end operation.

operation non_transactional_write(addr, value) is
(87)    store(addr,unMark(value))
end operation.
```

Fig. 8. Non-transactional operations for reading and writing a variable.

the write-set of the transaction, and that each transaction has a marker that is initialized as *LIVE* and is set to *COMMITTED* just before the transaction starts performing write backs during the commit phase. The NT-read operation is shown in Fig. 11.

```
operation transactional_read(addr) is
(88)    if addr ∈ ws then return (item.value from addr in ws) end if;
(89)    tmp ← load(addr);
(90)    if (tmp is of type T )
(91)      then if (status = LIVE) then abort() end if;
(92)           if (tmp.time > rv) then abort() end if;
(93)           if (status = COMMITTED)
(94)             then value ← tmp.val
(95)             else value ← tmp.last
(96)           end if;
(97)      else
        % Do validation to prevent abort due to a non-transactional write
(98)           rv ← validate_by_value();
(99)           value ← tmp;
(100)   end if;
(101)   if this is an update transaction then add value to rs end if;
(102)   return (value)
end operation.

operation transactional_write(addr, value) is
(103)   if addr ∉ ws
(104)     then allocate a new variable item of type T;
(105)          item ← (addr, value, status, ∞); ws ← ws ∪ item
(106)     else set item.value with addr in ws to value
(107)   end if
end operation.
```

Fig. 9. Transactional operations for reading and writing a variable.

```
operation try_to_commit() is
(108)   if (ws = ∅) then return (COMMITTED) end if;
(109)   for each (item ∈ ws) do
(110)     tmp ← load(addr);
(111)     if (tmp is of type T)
(112)           then if ((status ← tmp.status) = COMMITTED)
(113)                then item.last ← tmp.value
(114)                else if (status = ABORTED) then item.last ← tmp.last
(115)                else abort()
(116)              end if;
(117)           else item.last ← tmp
(118)     end if;
(119)     if (¬C&S(item.addr, tmp, item)) then abort() end if;
(120)   end for;
(121)   time ← fetch&increment(GVC);
(122)   validate_by_value();
        % Ensure the writes haven't been overwritten by non-transactional writes
(123)   for each (item ∈ ws) do
(124)     if (item ≠ load(item.addr)) then abort() end if
(125)     item.time ← time;
(126)   end for;
(127)   if (C&S(status, LIVE, COMMITTED))
(128)     then return (COMMITTED)
(129)     else abort()
(130)   end if;
end operation.
```

Fig. 10. Transaction commit.

```
operation non_transactional_read(addr) is
(131)   lock ← load_lock(addr);
(132)   value ← load(addr);
(133)   if ( lock is locked ∧tmp.status = COMMITTED ∧addr ∈ lock.ws)
(134)     then value ← item.value from addr in lock.ws
(135)   end if;
(136)   return (value)
end operation.

operation non_transactional_write(addr, value) is
(137)   Perform a transactional begin/write/commit operation
end operation.
```

Fig. 11. Non-transactional operations for reading and writing a variable.