Software Transactional Memory

Vladimir Sedach Andrew Seniuk

May 2, 2007

1 Introduction

Software transactional memory (STM) is a shared-memory concurrency model, originally inspired by cache coherency protocols in hardware design, but having the flavour of version control systems such as CVS. The main idea is that a process initiates a transaction which obtains a private copy of the data to be modified, does local computation, and when finished attempts to commit the results back to shared memory. The commit will succeed only if validation checks ascertain that the transaction has seen a consistent view of memory; otherwise it must retry. The transaction appears to execute atomically at some point in time within its execution, or in other words STM is *linearizable*. Although lock-based code tends to run more efficiently, the STM approach has appeal in that locks need not be used, so that sequential code can in most cases be safely converted to concurrent simply be wrapping code into modular, composable transactions.

2 Lock-based approaches to concurrency

Locks can be used to guard access to individual objects, critical sections of code, or even for some abstract mutual exclusion protocols (for example, to implement barriers). The main problems with locks from the point of view of programmer ease-of-use come from locking protocols and composability.

When a programmer wishes to act on multiple lock-protected objects, she has to know exactly which locks corresponding to which objects must be acquired. While constructs such as Java's synchronized methods alleviate this to some degree, they do not eliminate the need to know exact details of locking protocols entirely, and neither do they address the fact that a locking protocol consists of more than just the set of locks needed to ensure safety (lack of race conditions).

Consider the following pseudo-code:

```
P1:                                  P2:
acquire(lock1);                      acquire(lock2);
acquire(lock2);                      acquire(lock1);
...                                  ...
```

The example illustrates two locks being acquired in a different order by two different processes, resulting in a deadlock - the progress condition can no longer be satisfied and the two processes will wait on each other forever.

One solution to the above problem is to always ensure that a process acquires all of its locks in the same order as all other processes. The only way to do this is by implementing lock acquisition order in the program logic itself - this either places undue burden on the programmer, or, much more commonly, cannot be done due to the fact that the particular locks a process needs to acquire cannot be anticipated in advance when the shared objects manipulated by each process must be chosen at runtime.

Another solution is to use elaborate deadlock-detection algorithms to detect and break deadlock. Although conceptually an interesting approach that solves the problem, currently known deadlock detection algorithms impose very high runtime overhead and so are rarely used in practice.

The choice for ensuring progress (deadlock-freedom) for lock-based programs comes down to the inflexible (acquire all locks in the same order) or the impractical (deadlock detection algorithms). It is then no wonder that currently, concurrent programming is considered a very difficult subject by the majority of practicing programmers.

Composability of lock-based code breaks down because of the same need to know the details of the locking protocol of all objects that need to be acted upon. Say we developed a banking system where each account is represented by an object with synchronized withdraw and deposit methods. Since they are synchronized, withdraws and deposits happen atomically. Now we wish to implement transfers between accounts, which we would also like to be atomic. Obviously, `transfer(A,B,x)` { `withdraw(A, x); deposit(B, x);` } is not going to work. Now the locking protocol of the accounts will have to be exposed, so perhaps something like the following might work

```
transfer(A,B,x) {
   synchronized(A) {
      synchronized(B) {
         withdraw(A,x);
         deposit(B,x);
      }
   }
}
```

will work. However, what if there is a global `TotalBankBalance` object that holds the sum of the bank's account balances and is updated by the withdraw and deposit methods? If the programmer

wants transfers to be atomic, she would have to know about it and lock it as well. Any time the set of objects needing to be synchronized on in a particular method changes, all code that composes the method in an atomic way will have to update the set of locks it tries to acquire to reflect the change.

3 Focus on DSTM

The dynamic software transactional memory (DSTM) of Herlihy *et al.* is a system with both practical and theoretic appeal. Although a specific implementation with examples in Java are given in [3], we focus here on the model and proof of correctness, with some discussion of progress guarantees.

3.1 The DSTM Model

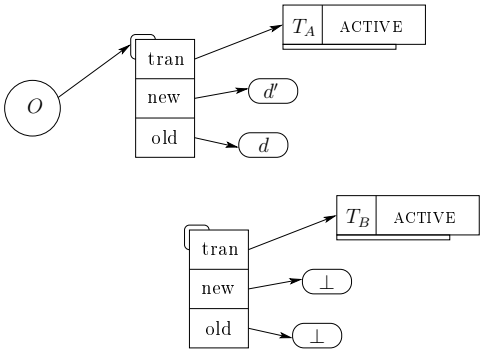
DSTM uses the Compare-and-Swap (CAS) to atomically swap locator pointers within objects, and to atomically change the status of a transaction. The objects are generic data objects (which must implement deep cloning), wrapped in a transactional memory object. Locators are records which are created any time a TM object is opened in WRITE mode. Any TM object points to at most one locator. A locator has three fields, represented by the triple $(\mathsf{t}, \mathsf{n}, \mathsf{o})$. t is a pointer to a transaction, and n and o are pointers to new and old copies of the raw data being modified (Figure 1). If the transaction[1] commits successfully, then the new version n becomes current and obtains system-wide visibility; if the commit fails, or t is aborted, then the old version o remains current and the modified version n is discarded.

If a transaction has modified multiple objects and succeeds, all those object locators' n fields become current atomically. This is possible by atomically swapping the status field of t from ACTIVE to COMMITTED. Thus is atomicity achieved at the granularity of transactions.

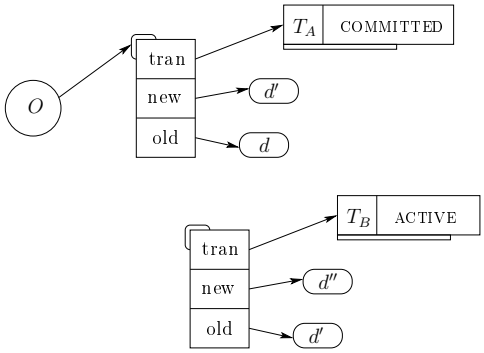
When a transaction B opens an object which is already open for writing by another transaction A, it cannot know which version, n or o, of the object's present locator will become current in the future. Thus it must either wait to learn the outcome of A, abort itself, or abort A. The only one of these three possibilities which is consistent with a non-blocking progress guarantee is the latter, so A will try to atomically swap the status of B to ABORTED in order that A may make progress. This is enough to guarantee obstruction freedom, although it does not qualify for lock-freedom. Note that B may not succeed in aborting A, since A may commit before causing the A's CAS on B's status to fail. This is even better, since now B *does* know that the new, modified version of the object is current, and can proceed to use it.

Something needs to be said about objects opened in READ mode, since it is the 'read-checks' associated with such openings that make the linearizability proof subtle. When a DSTM transaction B opens an object in READ mode which has been opened in WRITE mode by a concurrent

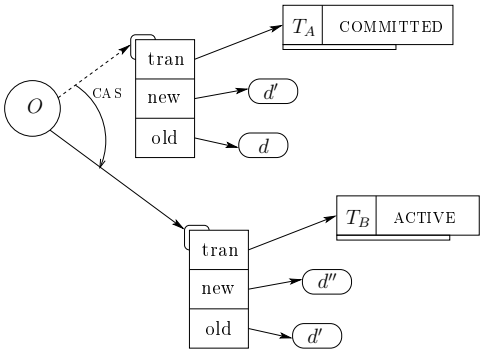
[1] We take a liberty and call t 'the transaction' although it is a pointer.



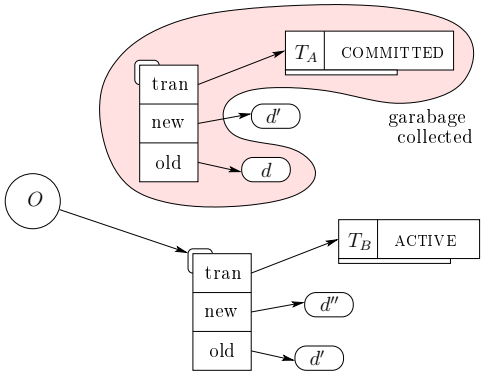
(a) T_B prepares an new locator, eventually for O.



(b) T_A commits successfully, so that d' is the current value of O. T_B prepares the `old` and `new` fields of its locator.



(c) T_B CASes O's locator pointer to its locator.



(d) d' is still currently referenced, but the rest of the old locator and what it points to is garabage.

Figure 1: Progress of two transactions with contention over object O. Consider also that between **(a)** and **(b)** T_B probably attempted to CAS the status of T_A to ABORTED, but failed because T_A committed in the interim.

transaction A, B has to go through the same process as when it opens in WRITE mode, except that when B finally obtains a current version of the data (by aborting A, or by A committing before it could be aborted), it doesn't create a new locator. Rather, it leaves A's old locator intact, but it makes an entry for the object in B's thread-local *read-only table*. Every transaction's entire read-only table is validated on every open operation, and at the beginning of the commit phase.

3.2 Linearizability

There are two basic approaches to proving linearizability, either to prove that any possible execution of the model admits a linearization, or to prove that there exists a fixed choice of linearization point for any operation such that any execution behaves as if all operations executed atomically at those points. It is the latter approach which is used here, as well as in other STM linearisability proofs with the possible exception of [9]. The DSTM is linearized at the start of the final read check (beginning of the commit phase). No correctness proof is given in [3], and we attempt to remedy the deficiency here.

We wish to prove that the DSTM implementation described in the last section preserves strict consistency – every read returns the value of the last write occurring in any successfully committed transaction. Let's establish some proof notation and formalize the problem.

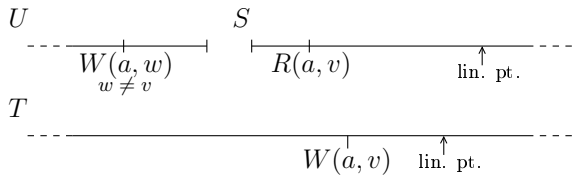
$T.X_i$	The ith operation of transaction T.[1]
$S.X \rightarrow T.Y$	Operation X in transaction S happens before Y in T.
$T.B$	Transaction T begins.
$T.E$	T ends (either by being aborted or by committing successfully).[2]
$T.C$	T performs a successful commit CAS, swapping it's own state from ACTIVE to COMMITTED.
$T.O^W(a)$	T opens TM object a in WRITE mode.
$T.W(a, v)$	T writes value v to (local copy of) object a.[3]
$T.O^R(a)$	T opens TM object a in READ mode.
$T.R(a, v)$	T reads value v for object a from T's read-only table.
$T.K(a)$	T performs read-check on object a.[2]
$T.K$	T performs read-check on every object in its the read-only table.

[1] X and Y stand for generic operations. The rest of the operation symbols are specific (and mnemonic).

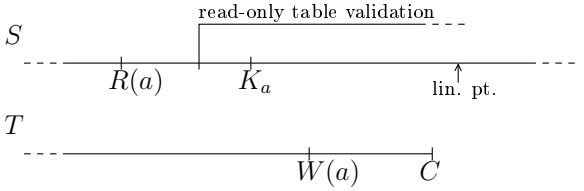
[2] We assume that a transaction which fails any read check self-aborts, although this is not made explicit in [3].

[3] We occasionally write $T.W(a)$ or $T.R(a)$ when we don't care about the value.

Formally, what we need to prove is that when the following scheme holds prior to linearization,



(a) Illustrating why you can't linearize DSTM prior to the read-only table validation at the start of the commit phase.



(b) Why the DSTM linearization point cannot be later than the start of the read-only table validation. In particular, DSTM cannot be linearized at the commit CAS, contrary to the claim in [6].

Figure 2: Counterexamples showing why the DSTM linearization point cannot be before or after the start of the commit-phase read-only table validation.

it remains valid afterwards.

$$\begin{aligned} & T.W_i(a,v) \rightarrow T.C \rightarrow S.R_j(a) \\ \wedge\neg\ & T.W_i(a,v) \rightarrow T.W(a) \\ \wedge\neg\ & T.W_i(a,v) \rightarrow U.W(a) \rightarrow U.C \rightarrow S.R_j(a) \\ \Rightarrow\ & S.R_j(a,v) \end{aligned}$$

To begin, consider why the DSTM linearization point (if one exists) could *not* be chosen at the final CAS of the commit. Figure 2a depicts a history which fails to linearize at an arbitrary point prior to the read-only table validation (commit read-check). The complementary Figure 2b shows why linearization can fail if the linearization point is chosen beyond the start of the read-only table validation.

For what follows it is useful to consider histories in which the same value is never written twice to the same TM object. While this is not necessarily the case in an arbitrary DSTM history, it suffices to limit our argument to this 'worst case', where validity is actually threatened. The advantage is that, if linearization reorders our ops we don't need to worry that the value read is still valid due to some unforseen prior write of the same value.

Lemma. $\neg\ (T \neq U\ \wedge\ \{T.W(a), U.W(a)\} \rightarrow \{T.C, U.C\})$.

Proof.

$$\begin{aligned} & \{T.W(a), U.W(a)\} \rightarrow \{T.C, U.C\} \\ \Rightarrow\ & \{T.O(a), U.O(a)\} \rightarrow \{T.C, U.C\} \\ \Rightarrow\ & \quad T.O^W(a) \rightarrow U.O^W(a) \rightarrow \{T.C, U.C\} \\ \vee\ & U.O^W(a) \rightarrow T.O^W(a) \rightarrow \{T.C, U.C\} \end{aligned}$$

both of which are contradictions. Consider w.l.o.g. the first disjunct. When U attempts to open the TM object a for writing, it will attempt to abort T. If T manages to commit before being

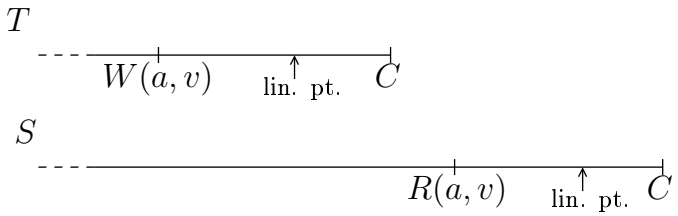


Figure 3: The only remaining possibility for a write and read to the same TM object. The extent of overlap between T and S could vary, but in any case the read in S must follow the commit in T. This linearizes correctly.

aborted, the above sequence does not reflect the actual order of ops, since $T.C \rightarrow U.O^W(a)$ in that case.

Theorem. *DSTM is linearizable.*

Proof. The possible history has been reduced to a situation like that shown in Figure 3. This linearizes correctly – if all operations were to occur at the linearization points shown, this would not affect the apparent ordering of write and read to a.

4 GHC STM

The GHC STM system, originally described in [2], is an implementation of software transactional memory for the Glasgow Haskell Compiler. STM operates on shared mutable places called TVars (short for Transactional Variables), which can hold values of any type (since Haskell is strictly typed[2], once a TVar is declared to be of a particular type, it can only hold values of that type).

4.1 GHC STM programming interface

Due to its monadic nature, the GHC STM implementation differs markedly from STM implementations for conventional imperative programming languages, and is well worth studying.

The basic shared object of STM is the Transactional Variable:

```
data TVar a
```

`TVar` is defined as a type that can "hold" any other type.

```
newTVar :: a -> STM (TVar a)
```

[2]The Haskell type system is actually based on Hindley-Milner types, a much more powerful and concise system than the ad-hoc strict/static type systems found in conventional imperative programming languages.

Given an argument of type `a`, `newTVar` returns an STM action that returns a new transactional variable which initially has the given argument as data.

```
readTVar :: TVar a -> STM a
```

Given a transactional variable as an argument, `readTVar` returns an STM monad that holds the contents of that transactional variable. The contents can be bound to a variable with the `<-` operator.

If we've created a `TVar` of type `a`, call it `A`, then `foo <- readTVar A` takes the type `STM a` (yielded by `readTVar`) to `a` and binds it to the variable `foo`.[3]

```
writeTVar :: TVar a -> a -> STM ()
```

Given a transactional variable and a new value for it, `writeTVar` creates an STM action that writes the new value to that transactional variable.

```
atomically :: STM a -> IO a
```

`atomically` takes an STM action (transaction) and returns an IO action that can be composed (sequenced) with other IO actions to build up a Haskell program.

The following code listing provides an example of how the above can be put together, as well as illustrating the use of higher-order programming with monads:

```
module Main where

import Control.Concurrent.STM
import Control.Concurrent

main = do { i <- getInt;
            a <- makeAccount 100;
            b <- makeAccount 200;
            atomically (do { withdraw a i; deposit b i });
            act <- printBalance a;
            act}

data Account = Balance (TVar Int)

makeAccount :: Int -> IO Account
```

[3]The `<-` operator is really syntactic sugar for the more general concept of monad binding.

```
makeAccount x = atomically (do { tvar <- newTVar x; return (Balance tvar) })

withdraw :: Account -> Int -> STM ()

withdraw (Balance balanceTVar) amount =
    do { balance <- readTVar balanceTVar;
         writeTVar balanceTVar (balance - amount) }

deposit (Balance balanceTVar) amount =
    do { balance <- readTVar balanceTVar;
         writeTVar balanceTVar (balance + amount) }

printBalance :: Account -> IO (IO ())

printBalance (Balance balanceTVar) =
    atomically ( do { balance <- readTVar balanceTVar;
                      return (print balance) } )

getInt = do { line <- getLine;
              return (read line :: Int) }
```

In addition to the usual transactional operations, GHC STM provides two operators that when taken together form a powerful new synchronization mechanism for transactional code:

```
retry :: STM a
```

`retry`, when called, causes the transaction to retry from the beginning. However, instead of running right away, the transaction will block on all the transactional variables that were previously accessed by that transaction until at least one of them is modified (since `TVar`s are the only stateful objects inside an STM monad, non-determinism/conditional execution inside transactions is entirely dependent upon them - if all the `TVar`s have the same values, re-running the transaction will lead down the same execution path that lead to the `retry`).

```
orElse :: STM a -> STM a -> STM a
```

`orElse` is a function that enables non-deterministic composition of transactions. `orElse` takes as arguments two STM actions, and returns an STM action that when executed will ensure that only one of the two sub-transactions will succeed. `action1 ‘orElse‘ action2` first executes `action1`, and if `action1` retries (via `retry`), then it executes `action2`. If `action2` also retries, `orElse` tries to run `action1` again, but first it blocks, waiting on changes to `TVar`s that were accessed by either of the two actions (for the same reason that `retry` blocks on `TVar`s). `orElse` is associative (in the sense that `a1 ‘orElse‘ (a2 ‘orElse‘ a3) = (a1 ‘orElse‘ a2) ‘orElse‘ a3`), and further its unit is `retry` (that is, `retry ‘orElse‘ action` and `action ‘orElse‘ retry` is the same as just `action`).

4.2 GHC STM semantics

Because Haskell's type system restricts side-effects to IO actions, the GHC Haskell STM system can be described with an operational semantics that largely corresponds to the Haskell code. In [2], Harris et alia provide such semantics. They will not be reproduced here, however, the authors feel that they warrant some comment.

The semantics are provided in terms of state transition rules corresponding to STM and IO actions. It is important to note that the semantics describe the results of the system interface only - most implementation details are left out. In particular, while the semantics specify that IO transitions can be interleaved between several threads, STM transitions transition to a return or throw statement instead of transition-at-a-time like IO transitions. So an STM action that has been wrapped in 'atomically' yields an atomic transition - in effect linearizability is already assumed.

An important property of the semantics is that they illuminate tricky design decisions with `orElse` and exceptions - if the first action throws an exception, should it be discarded and the second action attempted (which at first glance may seem like a reasonable thing to do), or should the exception propagate? If the former happens, then what about if the second action throws an exception? This can't be ignored (otherwise `orElse` would trap all exceptions!), so any exceptions thrown by `action2` would be propagated, but not those of `action1` - besides being inconsistent, this would also break the relation that `retry` is a unit of `orElse`.

Another *critically* important aspect of exception handling inside transactions is revealed in [2] that to the authors' knowledge has not been considered elsewhere. Software transactional memory systems with lazy acquire semantics (see Section 6) open the possibility that an exception can be thrown because an inconsistent state has been observed (this is not possible in eager acquire systems because validation occurs each time an object is opened for subsequent reading and/or writing). From a transactional point of view, the correct way to handle this situation is to catch the exception, validate the transaction, and if the validation fails, discard the exception and retry the transaction, and if it succeeds, re-throw the exception.[4] This is the approach that the GHC STM system employs. Precisely because side-effects are limited to transactional variables in the STM monad, this particular choice is guaranteed to be correct. However, in systems with unrestricted side-effects, the possibility arises that the above mechanism will discard valid exceptions carrying meaningful state. The authors feel that this issue can lead to potentially difficult to diagnose bugs, and the choice of exception handling semantics for lazy-acquire software transactional memory systems should be carefully considered and specified by the implementers of those systems as this affects the behavior of programs written by their users.

The biggest hole left from the viewpoint of someone attempting to prove correctness properties about the GHC STM system is the assumption made by the semantics that the STM transitions are already linearized. Of course, in order to prove this piece of the puzzle, we will need to

[4]Languages with less primitive condition handling facilities, such as Common Lisp, enable handling of these cases without needing to catch and re-throw. Since the exception system is implemented monadically in Haskell, it does not suffer from this weakness either.

dive down into the STM implementation code at its operating system interface. For portability, interfacing and performance reasons, much of the implementation code is written in C (as is much of the GHC multithreading system), which makes a formal proof of all but the most trivial properties of the source code nearly impossible. However, we can make an assumption that the implementation code is correct, and prove properties about the high-level description of the design of the implementation, like was done earlier in the paper for DSTM.

5 STMs Categorized by Definition of Conflict

Michael Scott in [8] makes several useful contributions to STM theory by offering an abstracted model about which it is simple to reason, and then applying this model to a classification of conflict (contention) which encompasses and contrasts the major implemented STM systems.[5]

The desire to abstract the essential STM model away from all accoutrements is a step in the same direction as taken by Herlihy *et at.* [3] when they chose to separate the contention manager from the STM proper. In their case, the separation was motivated principally by practical modularity considerations – the contention manager can be 'swapped' without changing the basic STM model, a feature which the ASTM system has capitalized on as we shall see in Section 6. Scott has been able to go further because he is not describing any implementation. His motive is mainly elucidatory, and makes it possible to prove linearizability of his abstract STM with a minimum of ado. However, the proof is not very interesting since he linearizes transactions at their terminal commit point, a nicety which is not possible in any of the real-world implementations we have examined. For this reason we chose to focus on linearizability of the DSTM, which offers more insights into the problem domain. On the other hand, the categorization of conflict was, we thought, refreshing and illuminating, so we precis it here.

Very briefly, the Scott abstract model of STM models a *transactional memory* (TM) as a mapping from state objects to values. Initially all state objects' values are $\perp$ (undefined). The TM is a high-level object that supports one type of operation, the transaction. A **transaction** is a sequence of low-level atomic operations ('ops') of the form

start (read|write)* (commit|abort)

performed by a single thread. Every op takes argument t, a unique **transaction descriptor**. read(o,t) returns the value of object o, or $\perp$ if o was never written, write(o,d,t) writes a value d to object o, and commit(t) returns **succeed** or **fail**.

The collection $\mathcal{H}$ of all valid histories of the TM are taken to be well-formed at the granularity of transactions, and sequential at the granularity of low-level operations. The sequential specification of the TM at bare minimum assumes that a commit of any isolated transaction will succeed, and that all read ops in a successfully committed transaction are *valid*, meaning that they return the

[5]He also develops a theory of arbitration to encompass contention management strategies but that is beyond our purview.

value of the most recent write which was previously successfully committed, and that this value is unchanged at the the commit of their own transaction. No ops occur outside of a transaction, and no transaction nesting within a thread is permitted.

A conflict function $C : \mathcal{H} \times \mathcal{D} \times \mathcal{D}$ is defined to be a boolean function over the product of histories and descriptor-pairs. Two transactions with descriptors s and t in a history H *conflict* precisely when $C(H, \mathsf{s}, \mathsf{t})$ holds true. The conflict functions are required to preserve three properties:

1. $C(H, \mathsf{s}, \mathsf{t}) = C(H, \mathsf{t}, \mathsf{s})$,
2. if $\mathsf{s} = \mathsf{t}$ or s and t refer to nonoverlapping transactions, then $C(H, \mathsf{s}, \mathsf{t})$ is false, and
3. if $H_{[\mathsf{s},\mathsf{t})} = I_{[\mathsf{s},\mathsf{t})}$ then $C(H, \mathsf{s}, \mathsf{t}) = C(I, \mathsf{s}, \mathsf{t})$, where $H_{[\mathsf{s},\mathsf{t})}$ denotes the subhistory of H consisting only of ops of s and t, and not including any ops beyond the end of s or t.

The first property implies that it is the pair (s, t) that conflicts in H, and that neither s nor t is the cause of it. (Scott introduces *arbitration functions* to break this symmetry, for contention management, but we cannot elaborate on that here.) The second property upholds our sequential specification in that isolated transactions must commit successfully. The third property means that other transactions don't influence the conflict valuation between s and t.

Figure 4c illustrates the nested structure of STM models which is induced by some natural alternative conflict definitions. The regions can be thought of as subspaces of $\mathcal{H} \times \mathcal{D} \times \mathcal{D}$ – specifically, inverse images of true, $C^{-1}(\mathit{true})$. Thus, the enclosing conflict function is less permissive than the enclosed, so that 'overlap conflict' is the most stringent, and 'lazy invalidation' the most permissive.

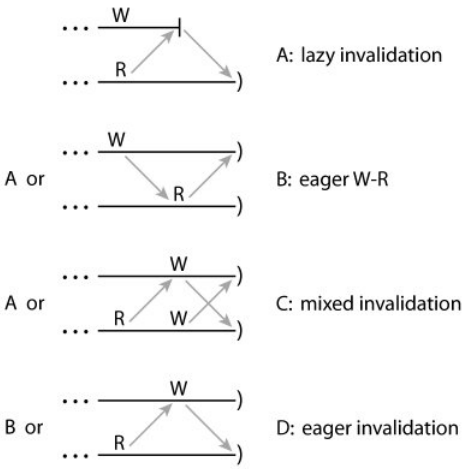
Lazy invalidation defines there to be a conflict if in two overlapping transactions s and t, s writes to some object that t reads, and s commits successfully before t finishes. Lazy invalidation conflict is the most permissive consistency-preserving definition of conflict, by definition, since if a conflict was not generated by this situation, the value read in t would be invalid for the duration of t after the commit of s. It is instructive to examine how the conflict functions differ; as an example, consider now 'eager W-R' invalidation (Figure 4a). Unlike lazy invalidation, eager W-R requires that the write in s happens before the read in t, but it does not require that either s or t commit for a conflict to arise. One says that the write in s 'threatens' the read in t, because there exists a possible future history in which the read is invalidated by the write, namely a history in which s commits first.

Let's leave Scott by proving one of his main results, which requires one more definition. The authors regret that this introduces two more undefined terms, but if all the terminology was to be defined this section would be as long as Scott's entire paper! The terms in question will be defined informally in the course of the following proof.

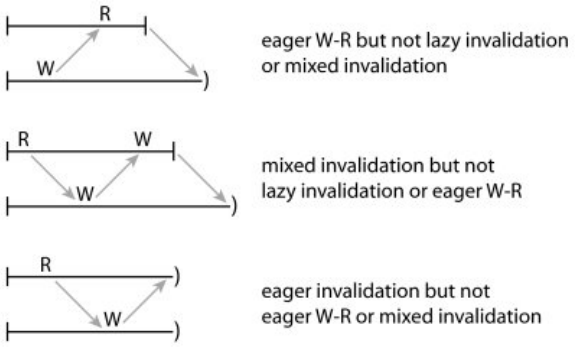
Theorem. *For any conflict function C, C-based TM is a sequential specification.*

Proof. *C-based TM* denotes the set of all consistent, C-respecting histories. A sequential specification is a prefix-closed set of sequential histories. A C-respecting history is one in which both

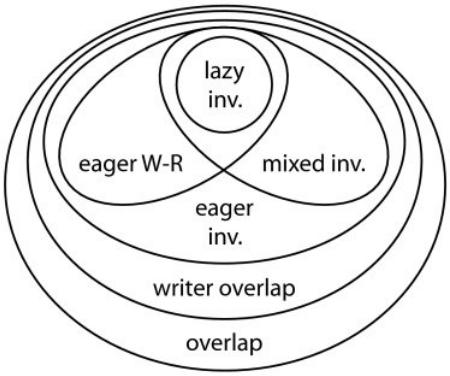
1. it is never the case that both of a conflicting pair of transactions commit, and



(a) Some nontrivial conflict definitions.



(b) Demonstrating nondegeneracy in the definitions.



(c) A hierarchy of conflict.

Figure 4: A classification of STM systems by definition of conflict. Diagrams reproduced from [8].

2. any transaction which has no conflicts succeeds.

The properties of a conflict function imply that, if a history is C-respecting, it is linearisable, by Theorem 1 of [8]. It only remains to prove that C-based TM is prefix-closed. Supposing the contrary, let P be a prefix of a C-respecting history H, such that P is not C-respecting. We take the two cases of the definition of C-respecting in order.

In the first case, there must exist two C-conflicting transactions S and T in P which both commit successfully in P. But since P is a subhistory of H and commits are irrevocable, these two commits also succeed in H. Also, the conflicting transactions must still conflict in H by property 3 of the definition of conflict function, since $P_{[S,T)} = H_{[S,T)}$. This is a contradiction, since H could not be C-respecting under these conditions.

In the second case, there must exist a failed transaction T which had a conflict (say with transaction S) in P but none in H. Since T failed in P, it must have ended in P, which means again that $P_{[S,T)} = H_{[S,T)}$. That would imply that $C(H, S, T) = C(P, S, T)$, a contradiction. □

6 Survey of Preeminent STM Systems

The paper of Marathe *et al.* [6] introducing adaptive STM (ASTM) is generally a useful guide to the comparative anatomy of the foremost published STM systems, with Scott [8] providing additional insights. We will consider the DSTM system [3] of Herlihy *et al.* (recall Section 3) and the OSTM system [1] of Fraser and Harris, as well as the ASTM which is an compromise between these.

Following Maranthe, the design space of STM can be usefully modelled as a 2^4-space:

1. acquire semantics (eager *vs.* lazy)
2. metadata structure (per-object *vs.* per-transaction)
3. indirection in object referencing (direct *vs.* indirect)
4. non-blocking progress guarantee (obstruction-free *vs.* lock-free)

Of course there are more than sixteen possible STM systems, but these categories do capture some essential characteristics. A word about the meaning of aquire is in order. Recall that when an object is opened for writing in DSTM, it is aquired in the sense that the TM object's locator pointer must be CASed from the locator of the currently owning transaction to the new transaction. If this is done eagerly, obstruction-freedom is the strongest possible progress guarantee. If aquire is postponed, it is possible to achieve lock-freedom. Generally, it is impossible to know which of two conflicting transactions would succeed, or would succeed soonest, if the other were aborted, so lazy aquire semantics has an appeal on those grounds. However, Maranthe *et al.* report that performance evaluation shows no significant advantage to either strategy.

DSTM is an eager-acquire, indirect object referencing, per-object metadata, obstruction-free STM. OSTM is at the opposite corner of the cube, with lazy-acquire, direct object referencing, per-transaction metadata, and lock-free progress. ASTM is adaptive in the sense that it switches between modified forms of DSTM and OSTM depending on the nature of the workload. In a

nutshell OSTM is the winner in read-dominated tasks (due to the lack of indirection), but is the slower in write-dominated tasks (due to the extra overhead of validating writers at every open() op, whereas DSTM only requires that readers be validated).

Like DSTM, ASTM (both eager and lazy variants) maintain a read-list which is validated on each open() op, but lazy ASTM goes further and never attempts to acquire any object until commit time, instead maintaining an additional write-list analogous to the read-list. This does not make the ASTM lock-free however, as it still relies on a contention manager to resolve conflicts. The OSTM achieves lock-free progress by maintaining a sorted order of the objects, and using *recursive helping* to allow transactions to expedite the task completion of contending transactions, a technique which has been used in STM systems since Shavit and Touitou [9]. ASTM also incorporates *early release* features found in DSTM, allowing readers which are no longer needed in a transaction to erase their entries from the read-table, a practise which can decrease the validation complexity from quadratic to linear. This is especially useful for structures which need to be traversed from a common ingress, such as trees and linked lists. However, it has the disadvantage of breaking linearizability! The system is no longer safe; it becomes incumbent on the programmer to exercise correct judgement in deciding when it is safe to release an object. The OSTM makes validation checks the responsibility of the programmer *a priori*, and therefore suffers in this respect also. Under the assumption that all necessary validation checks are in place, both OSTM and DSTM linearize at the beginning of the commit phase, before the read-check validation. (The linearization point is incorrectly reported in [6] to be the final CAS which effects the commit.)

In terms of Marathe et alia's STM design space, GHC STM is a lazy acquiring, per-transaction metadata, indirect object referencing, and lock-free (a transaction is only aborted if another conflicting transaction was the first to commit).

7 Software transactional memory subtleties and pitfalls

While software transactional memory offers a promising alternative to explicit locking protocols, transactions are not a drop-in replacement for locks. Problems also exist in the interaction of transactional and non-transactional code.[6] The main source of these problems stems from the fact that while locks can be acquired per-object, transactions make the atomicity condition apply to *all* objects in their scope.

The following is an example (borrowed from [7]) of a busy-waiting barrier inside critical sections where only two different objects need to be locked, that would work correctly in the presence of an explicit locking protocol, but deadlocks when naively converted to transactional code:

```
boolean flagA := false;
boolean flagB := false;
Object o1, o2;
```

[6] This is obviously a non-issue for the GHC STM system.

```
// this works

P1:                                     P2:
synchronize(o1) {                       synchronize(o2) {
        while (!flagA) {}                       flagA := true;
        flagB := true;                          while (!flagB) {}
}                                       }

// this will deadlock

boolean flagA := false;
boolean flagB := false;

atomic {                                atomic {
        while (!flagA) {}                       flagA := true;
        flagB := true;                          while (!flagB) {}
}                                       }
```

It is also important to note that the composability of transactions poses its own pitfalls. In particular, composition of transactions does not preserve the progress property of otherwise correct transactional code. In the following example (also borrowed from [7]) sequential composition leads to deadlock:

```
int A := 0;
int B := 0;

// this works correctly

P1:                                     P2:
atomic {                                atomic {
  A := 1;                                 if (A != 1) then retry
}                                         else B := 1;
                                        }
atomic {
  if (B != 1) then retry
  else...
}


// this deadlocks
```

```
int A := 0;
int B := 0;

P1:
atomic {
  atomic {
    A := 1;
  }

  atomic {
    if (B != 1) then retry
    else...
  }
}
```

```
P2:
atomic {
  if (A != 1) then retry
  else B := 1;
}
```

References

[1] K. Fraser and T. Harris. Concurrent programming without locks, 2007.

[2] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[3] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures, 2003.

[4] C. Manovit, S. Hangal, H. Chafi, A. McDonald, C. Kozyrakis, and K. Olukotun. Testing implementations of transactional memory. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 134–143, New York, NY, USA, 2006. ACM Press.

[5] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Design tradeoffs in modern software transactional memory systems. In *LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pages 1–7, New York, NY, USA, 2004. ACM Press.

[6] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sep 2005. Earlier but expanded version available as TR 868, University of Rochester Computer Science Dept., May2005.

[7] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2), 2006.

[8] M. L. Scott. Sequential specification of transactional memory semantics. In *ACM SIGPLAN Workshop on Transactional Computing*. Jun 2006. Held in conjunction with PLDI 2006.

[9] N. Shavit and D. Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.