Sequential Specification of Transactional Memory Semantics *

Michael L. Scott
Department of Computer Science
University of Rochester
scott@cs.rochester.edu

Abstract

Transactional memory (TM) provides a general-purpose mechanism with which to construct concurrent objects. Transactional memory can also be thought of *as* a concurrent object, but its semantics are less clear than those of the objects typically constructed on top of it. In particular, commit operations in a transactional memory may *fail* when transactions conflict. Under what circumstances, exactly, is such behavior permissible?

We offer candidate sequential specifications to capture the semantics of transactional memory. In all cases, we require that reads return consistent values in any transaction that succeeds. Each specification embodies a *conflict function*, which specifies when two transactions cannot both succeed. Optionally, a specification may also embody an *arbitration function*, which specifies which of two conflicting transactions must fail. In the terminology of the STM literature, arbitration functions correspond to the concept of *contention management*.

We identify TM implementations from the literature corresponding to several specific conflict and arbitration functions. We note that the specifications facilitate not only correctness (i.e., linearizability) proofs for nonblocking TM implementations, but also formal comparisons of the degree to which different implementations admit inter-transaction concurrency. In at least one case—eager detection of write-write conflicts and lazy detection of read-write conflicts—the formalization exercise has led us to semantics that are arguably desirable, but not, to the best of our knowledge, provided by any current TM system.

1. Modeling STM

We can model a *transactional memory* as a mapping from objects to values. Initially all values are undefined. The memory supports the following operations:

start(t) Begin transaction t. No return value.

read(o, t) Return the current value of object o in the context of transaction t. Return the distinguished value $\perp$ if o is uninitialized.

write(o, d, t) Write d to o in the context of transaction t. No return value.

commit(t) Attempt to commit transaction t and return a Boolean indication of success. The call is said to *succeed* iff it returns true.

abort(t) Abandon transaction t. No return value.

These definitions are intended to simplify correctness arguments, not to simplify programming. The richer interfaces typical of object-oriented software TM can be implemented in terms of these more basic primitives, without changing the underlying semantics. We defer discussion of such interfaces to Section 6.

Following the terminology of Herlihy and Wing [8], a *history* is a finite sequence of operation invocation and response events, each of which is tagged with its arguments and return values, and with the id of the calling thread. In a *sequential* history, each invocation is immediately followed by its matching response, with no events in between. A sequential history H thus induces a total order $<_H$ on its operations. Throughout the rest of the paper we will consider only sequential histories. We define the semantics of transactional memory on these histories.

A *transaction* is a sequence of operations, performed by a single thread, of the form (start (read | write)* (commit | abort)), where t is a unique *transaction descriptor* passed to start, to the commit or abort, and to every read or write in between. Transactions S and T in history H are said to *overlap* if $\mathsf{start}_S <_H \mathsf{end}_T$ and $\mathsf{start}_T <_H \mathsf{end}_S$, where end_T is T's commit or abort operation. Transaction T is said to be *isolated* in H if for all transactions $S \neq T$ in H, S and T do not overlap. We say a history H is *serial* if it consists of a sequence of isolated transactions, optionally followed by a single *uncompleted* transaction (i.e., a transaction prefix). For convenience, we associate end_T with the end of H if T is uncompleted (i.e., all operations in H precede the end of an uncompleted transaction). If S and T are both uncompleted, end_S and end_T are incomparable under $<_H$.

We assume throughout this note that all histories are *well-formed*, meaning that every thread subhistory is serial (we do not currently consider nested or overlapped transactions within a single thread). Well-formedness implies, among other things, a one-one correspondence between transactions and their descriptors. We also assume, for simplicity, that write is called no more than once for a given object within a given transaction. A transaction is said to *succeed* it if ends with a commit that succeeds. It is said to *fail* it if ends with a commit that fails. We use successful(H) to represent the history obtained by deleting from H all operations of failed, aborted, or uncompleted transactions.

As defined by Herlihy and Wing, a *sequential specification* $\mathcal{S}$ of a concurrent object O is a prefix-closed set of sequential histories on O. For most kinds of objects it is intuitively clear which histories should be in $\mathcal{S}$. Intuition is less clear for transactional memory. Certainly we must insist that reads return the "right" value in any transaction that succeeds. It also seems reasonable, at least in a

* Presented at TRANSACT: the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, held in conjunction with PLDI, Ottawa, Ontario, Canada, June 2006.

This work was supported in part by NSF grants CCR-0204344 and CNS-0411127, financial and equipment support from Sun Microsystems Laboratories, and financial support from Intel.

preliminary study, to insist that a commit succeed if it ends an isolated transaction. But under what circumstances may a commit operation fail?

To answer this question we first define, in Sections 2 and 3, a sequential specification that embodies the two minimal requirements just suggested. Our definition is driven by the notion of a *conflict function*, which specifies the circumstances in which two transactions cannot both succeed. In Section 4 we introduce a variety of conflict functions, leading to a rich structure of sequential specifications, several of which capture the semantics of published TM systems. We also identify an arguably attractive sequential specification that is not, to our knowledge, embodied in any published system. In Section 5, we consider the notions of *blocking* and *livelock*, and the extent to which they may be permitted or precluded by a sequential specification of TM. In particular, we introduce the notion of an *arbitration function*, which specifies, when two transactions conflict, *which* of them must fail. Section 6 explains how our model can accommodate an object-oriented API. We conclude in Section 7 with a summary and a list of open questions.

2. Consistency

We say a read operation $r = \mathsf{read}(\mathsf{o}, \mathsf{t})$ in history H is ***consistent*** if it returns the most recent committed value of o; that is, r returns d if there exists an operation $w = \mathsf{write}(\mathsf{o}, \mathsf{d}, \mathsf{s})$ in a successful transaction S such that (1) $\mathsf{s} \neq \mathsf{t}$, (2) $\mathsf{commit}_S <_H r$, and (3) for all operations $x = \mathsf{write}(\mathsf{o}, \mathsf{e}, \mathsf{u})$ in transactions $U \neq S$, if U is successful, then $\mathsf{commit}_U <_H \mathsf{commit}_S$ or $r <_H \mathsf{commit}_U$. If there is no such w, then o is uninitialized, and r returns $\perp$. Our definition does not make writes in T visible to subsequent reads in T, but this restriction is easily relaxed at a higher level of abstraction (we do so in Section 6).

We say a *history* H is consistent if (1) every read in every successful transaction is consistent, and (2) every such read is still ***valid*** when its transaction commits; that is, if $r = \mathsf{read}(\mathsf{o}, \mathsf{t})$ appears in a successful transaction T, then there exists an operation $w = \mathsf{write}(\mathsf{o}, \mathsf{d}, \mathsf{s})$ in a successful transaction S such that (a) r returns d, (b) $\mathsf{s} \neq \mathsf{t}$, (c) $\mathsf{commit}_S <_H \mathsf{commit}_T$, and (d) for all operations $x = \mathsf{write}(\mathsf{o}, \mathsf{e}, \mathsf{u})$ in transactions $U \notin \{S, T\}$, if U is successful, then $\mathsf{commit}_U <_H \mathsf{commit}_S$ or $\mathsf{commit}_T <_H \mathsf{commit}_U$. Note that this definition permits an implementation to ignore the ABA problem: a read is still considered valid at commit time if its value has been overwritten and then restored.

Lemma 1. In any consistent history, all reads of the same object in the same successful transaction return the same value.

Proof: Immediate consequence of the validity of reads. □

Theorem 1 (Fundamental theorem of TM). If H is a consistent history, then so is the serial history J consisting of all and only the transactions in successful(H), ordered according to the order of their commit operations in H.

Proof: Consider history I = successful(H). Clearly I is consistent, since the definition of consistency makes no reference to unsuccessful transactions. Now consider serial history J, consisting of all transactions of I, ordered according to the order of their commit operations in I. All of J's transactions remain successful, and its commit operations appear in the same order they did in I. Moreover because I's reads are valid at commit time, they remain consistent in J. Thus J as a whole is consistent. □

In the terminology of the database community [11, Sections 16.3 and 17.1], any history in which all reads are consistent *avoids cascading aborts*: when a transaction fails or aborts, an implementation never has to cause other transactions to fail in order to ensure consistency. Theorem 1, moreover, is equivalent to saying that consistent histories are *strictly serializable* or, equivalently, *linearizable* (since we never consider more than a single concurrent object—the transactional memory itself) [8]. There exist more relaxed notions of consistency in which transactions can read stale values that force them to "commit in the past" or, conversely, read speculative values from writes that have not yet been committed; we do not consider such extensions here.

3. Conflict

Consistency alone does not capture intuition regarding transactional semantics. A history in which no transaction ever succeeds is certainly consistent, but the set of all such histories is not an appealing sequential specification. It seems reasonable to require a commit operation to succeed unless its transaction T conflicts with some other transaction S, in which case at most one of them can succeed.

Let $\mathcal{H}$ be the set of all (well-formed) histories, $\mathcal{D}$ be the set of all transaction descriptors, and $H_{[\mathsf{s},\mathsf{t})}$ be the history obtained by removing from H all operations that specify a transaction descriptor other than s or t, or that follow commit(t), abort(t), commit(s), or abort(s) in H. (The notation is meant to suggest a half-open interval: $H_{[\mathsf{s},\mathsf{t})}$ includes the initial portions of both s's and t's transactions, but is missing a suffix of the one that finishes last.) A ***conflict function*** C is then a mapping from $\mathcal{H} \times \mathcal{D} \times \mathcal{D}$ to $\{\mathsf{true}, \mathsf{false}\}$ such that (1) $C(H, \mathsf{s}, \mathsf{t}) = C(H, \mathsf{t}, \mathsf{s})$; (2) if $\mathsf{s} = \mathsf{t}$ or if the transactions corresponding to s and t do not overlap, then $C(H, \mathsf{s}, \mathsf{t}) = \mathsf{false}$; and (3) if $H_{[\mathsf{s},\mathsf{t})} = I_{[\mathsf{s},\mathsf{t})}$, then $C(H, \mathsf{s}, \mathsf{t}) = C(I, \mathsf{s}, \mathsf{t})$. In other words, for overlapping transactions S and T, C makes its decision solely on the basis of the operations of those two transactions (and their interleaving) prior to the earlier of end_S and end_T.

For convenience, we use $H_{[S,T)}$ and $C(H, S, T)$ as shorthand for $H_{[\mathsf{s},\mathsf{t})}$ and $C(H, \mathsf{s}, \mathsf{t})$, respectively, where s and t are the descriptors of S and T, respectively. If $C(H, S, T) = \mathsf{true}$, we also say that "$S$ and T have a C conflict."

Lemma 2. Given any conflict function C, history H, and isolated transaction T in H, there is no transaction S that conflicts with T.

Proof: Immediate consequence of the definition of conflict. □

A history H is said to be ***C-respecting***, for some conflict function C, if (1) for every pair of transactions S and T in H, if $C(H, S, T) = \mathsf{true}$, then at most one of S and T succeeds; and (2) for every transaction T in H, if T ends with a commit operation, then that operation succeeds unless there exists a transaction S in H such that $C(H, S, T) = \mathsf{true}$. Put another way, if there is no S that conflicts with T, then T's commit succeeds.

For any given function C, we use the term ***C-based transactional memory*** to denote the set of all consistent, C-respecting histories. It seems reasonable to define conflict functions in a way that forces any C-respecting history to be consistent, but nothing about the definition of conflict requires this. We say that C is ***validity-ensuring*** if $C(H, S, T) = \mathsf{true}$ whenever there exists an object o and operations $r = \mathsf{read}(\mathsf{o}, \mathsf{t})$ in T and $w = \mathsf{write}(\mathsf{o}, \mathsf{d}, \mathsf{s})$ in S such that S ends with a commit and $r <_H \mathsf{commit}_S <_H \mathsf{end}_T$.

Lemma 3. If C is a validity-ensuring conflict function and H is a C-respecting history in which every read is consistent, then H is a consistent history.

Proof: Immediate consequence of definitions. □

Given the ABA problem, a validity-ensuring conflict function is sufficient but not necessary to ensure that all reads in successful transactions are still valid at commit time.

Perhaps the simplest conflict function is the following:

Overlap conflict: Transactions S and T in history H conflict if S and T overlap. *Overlap-based TM* thus consists of all histories in which every isolated transaction is successful and no two overlapping transactions are both successful.

Lemma 4. For any conflict function C, history H, and transactions S and T in H, if S and T have a C conflict, they also have an overlap conflict.

Proof: Immediate consequence of the definition of conflict function. □

Theorem 2. For any conflict function C, C-based TM is a sequential specification.

Proof: By the definition of sequential specification, we need only show that C-based TM is prefix-closed. Suppose the contrary: there exists some history $H \in C$-based TM and some H prefix $P \notin C$-based TM. There are two cases to consider. First, suppose there exist two successful transactions S and T that conflict in P but not in H. Since T is successful in P, P must include commit_T, which implies that $P_{[S,T)} = H_{[S,T)}$. But this implies that $C(P,S,T) = C(H,S,T)$, a contradiction. Second, suppose there exists some failed transaction T that has an excuse to fail in H but not in P. There must exist some transaction S in H such that $C(H,S,T) = \mathsf{true}$ but $C(P,S,T) = \mathsf{false}$. Since T fails in P, P must include commit_T, which implies that $P_{[S,T)} = H_{[S,T)}$. But this implies that $C(P,S,T) = C(H,S,T)$, a contradiction. □

4. Requiring concurrency

Overlap-based TM is a very weak specification; it admits an implementation in which overlapping transactions are never successful. An implementation might, for example, employ global counts of the number of started and active transactions. Operation start(t) would increment both counts and remember the started count; commit(t) would decrement the active count and return true iff the result were zero and the started count were equal to the remembered value.

To require that certain non-isolated transactions succeed, we must refine our definition of conflict, so more transactions are seen to be conflict-free. As a first step, we might insist that readers be permitted to proceed concurrently. (Remember here that we are still talking about sequential histories. Our goal is to increase concurrency among transactions, not [in this note] among individual operations.)

Writer overlap conflict: Transactions S and T conflict in history H if they overlap and one performs a write before the other ends.

Most TM systems go further, allowing transactions to proceed concurrently if they do not perform conflicting accesses to the same object:

Lazy invalidation conflict: Transactions S and T conflict in history H if there exist operations $r = \mathsf{read(o,\ t)}$ in T and $w = \mathsf{write(o,\ d,\ s)}$ in S such that S ends with a commit operation and $r <_H \mathsf{commit}_S <_H \mathsf{end}_T$. In other words, S and T conflict if S attempts to commit, and allowing it to succeed would invalidate a read in T.

Eager W-R conflict: Transactions S and T conflict in history H if (1) S and T have a lazy invalidation conflict or (2) there exist operations $r = \mathsf{read(o,\ t)}$ in T and $w = \mathsf{write(o,\ d,\ s)}$ in S such that $w <_H r <_H \mathsf{end}_S$. In other words, beyond the requirements of lazy invalidation conflicts, S and T conflict if a read in T is "threatened" by a previous write in S; that is, if w precedes r and the prefix of H that ends at r can be extended to create a history in which r is invalidated by w.

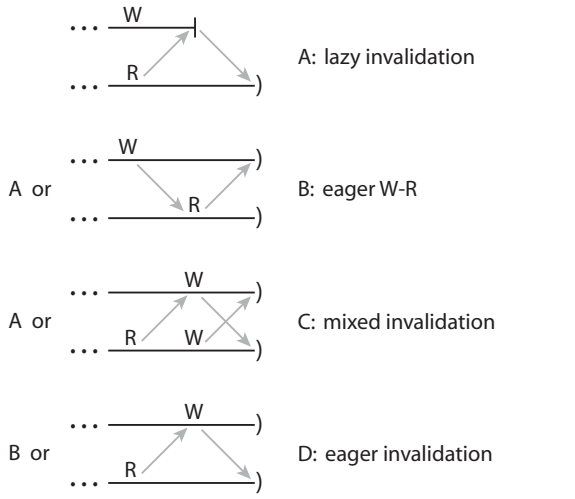

Figure 1. Alternative definitions of conflict. Arrows indicate history order. A straight terminator indicates a commit operation. A curved terminator indicates that a transaction may optionally be uncompleted.

Eager invalidation conflict: Transactions S and T conflict in history H if (1) S and T have an eager W-R conflict or (2) there exist operations $r = \mathsf{read(o,t)}$ in T and $w = \mathsf{write(o,\ d,\ s)}$ in S such that $r <_H w <_H \mathsf{end}_T$. In other words, beyond the requirements of eager W-R conflicts, S and T conflict if a read in T is threatened by a subsequent write in S; that is, if w follows r and the prefix of H that ends at w can be extended to create a history in which r is invalidated by w.

These definitions of conflict are illustrated graphically in Figure 1. None of them defines writes to the same object as conflicting: writes do not become visible to other transactions until commit time, and the fact that some other transaction is planning to update an object at some point in the future is harmless. Of course if a transaction *updates* an object—reading its value before writing it—then a concurrent write is indeed a conflict. Under the object-oriented API of Section 6, every write will be an update.

Note the asymmetry of eager W-R conflict: w would also threaten r if $r <_H w <_H \mathsf{end}_T$, but we do not define this as a conflict. The rationale for this asymmetry is that in a practical implementation a transaction must detect conflict with *previous* activity in some other transaction. The "other half" of eager invalidation, shown in Figure 1D, requires that readers be visible to writers. In practice, this in turn requires that readers modify some sort of metadata, inducing cache conflicts among readers that would not otherwise occur.

Lemma 5. Lazy invalidation conflict is the weakest consistency-ensuring conflict function.

Proof: Immediate consequence of definitions.

Claim (Proof omitted)**.** The OSTM of Harris and Fraser [1], with appropriate API adjustments (see Section 6) is an implementation of lazy invalidation-based TM. The DSTM of Herlihy et al. [7], with appropriate API adjustments and visible readers, is an implementation of eager invalidation-based TM. If it were augmented to permit validation of reads whose objects were subsequently acquired by not-yet-committed writers (our group refers to this as

"validating through"), DSTM with invisible readers would be an implementation of eager W-R-based TM.[1]

Note that the sets of histories induced by different conflict functions are generally incomparable. Consider, for example, the sequence of operations start(s) start(t) write(o, d, t) read(o, s) commit(s) commit(t). If this sequence is executed in isolation, the read must return $\perp$. The return values of the commits, however, will depend on the choice of conflict function: the transactions with descriptors s and t have an eager W-R conflict, but not a lazy invalidation conflict. The set of all lazy invalidation-respecting histories will include exactly one history corresponding to this sequence of operations: one in which both commits return true. The set of all eager W-R-respecting histories will include one in which both commits fail and two in which one succeeds but the other fails.

Eager W-R conflict gives transactions more excuses to fail than lazy invalidation conflict does (and eager invalidation conflict gives still more). In a practical implementation these extra excuses may or may not be a good thing. They are good if they allow the implementation to improve performance by heuristically abandoning work on transactions that are likely to fail (but see Section 5 below); they are bad if they allow the implementation to neglect opportunities for parallel speedup.

An implementation that uses a hash function h to locate transaction metadata might introduce the notion of h-*conflicting* transactions—transactions that perform conflicting accesses to objects in the same hash-induced equivalence class. Given a function h, assume some arbitrary total order on objects, and let let $g(\mathsf{a})$, for any object a, be the smallest object b such that $h(\mathsf{a}) = h(\mathsf{b})$. Then for any conflict function C, history H, and transactions S and T in H, S and T would be said to have an hC conflict if the transactions S' and T' have a C conflict, where S' and T' are obtained from S and T by replacing every object o in a read or write operation with its image $g(\mathsf{o})$. Definitions of hC-respecting histories and hC-based TM would follow accordingly.

Claim (Proof omitted)**.** The WSTM of Harris and Fraser [5] is an implementation of h-lazy invalidation-based TM for some appropriate hash function h.

If overlapping transactions S and T both read and then write the same object o, the argument for allowing S and T to proceed concurrently (as lazy invalidation does) is that any history in which both are uncompleted can be extended to abort either and commit the other; there is no way for an implementation to tell, *a priori*, which transaction "ought" to fail. This is a weak argument, however, since S and T cannot both succeed.

If, however, one of S and T writes o but the other merely reads it, there is a stronger argument for allowing them to proceed concurrently: both can succeed if the writer commits last. To capture this form of concurrency we can define the following:

Mixed invalidation conflict: Transactions S and T conflict in history H if (1) S and T have a lazy invalidation conflict or (2) there exist operations $r = \mathsf{read(o, s)}$ in S, $w_S = \mathsf{write(o, d, s)}$ in S, and $w_T = \mathsf{write(o, e, t)}$ in T such that $r <_H w_S <_H \mathsf{end}_T$ and $r <_H w_T <_H \mathsf{end}_S$. In other words, beyond the requirements of lazy invalidation conflicts, S and T conflict if (a) a read in T is threatened by a subsequent write in S, (b) the read is followed by a write in T, and (c) both writes happen before either transaction ends.

[1] As implemented, DSTM with invisible readers realizes semantics only subtly different from eager invalidation conflict: it admits histories in which both S and T are uncompleted, the last operation in T reads some object o, and there is a subsequent write of o in S.

Mixed invalidation conflict falls between lazy invalidation conflict and eager invalidation conflict, but is incomparable to eager W-R conflict. More formally and completely:

Theorem 3. The sets of transactions that have lazy invalidation, eager W-R, eager invalidation, and mixed invalidation conflicts are nested as shown on the left side of Figure 2, with each of the containments non-trivial.

Proof: Simple containment is an immediate consequence of the definitions of the respective conflict functions. Proper containment is illustrated by the examples on the right side of Figure 2. □

We are currently experimenting with mixed invalidation-respecting histories in our RSTM system [10]. To the best of our knowledge, no other existing system currently implements these semantics (without also being eager W-R-respecting).

5. Progress and arbitration

So far our discussion has addressed only *correctness*: what are the legal histories that may be realized by an implementation? One is also usually interested in *progress*: under what circumstances, if any, may a thread be blocked by the state of other threads? Traditionally progress has been discussed in the context of concurrent histories: when, if ever, can the response to an invocation be arbitrarily delayed? For transactional memory, however, we may also be interested in transaction-level progress in sequential histories: when, if ever, can a thread suffer an arbitrarily long string of failed transactions?

Consider, for example, the trivial implementation of overlap-based TM mentioned at the beginning of Section 4. This implementation clearly admits blocking at the level of transactions: given any history H in which transaction T is uncompleted, any extension of H in which T remains uncompleted will contain no successful transactions beyond the end of H. The implementation also admits livelock: we can easily construct a history in which every thread performs an arbitrary number of commits, none of which succeeds.

We define these conditions in the usual way:

Starvation: A sequential specification $\mathcal{S}$ is said to be *starvation-free* if for any thread a and any history H in $\mathcal{S}$ there exists an $n > 0$ such that in any H extension $H' \in \mathcal{S}$, if a performs more than n commit operations in H' after H, at least one of them will succeed.

Livelock: A sequential specification $\mathcal{S}$ is said to be *livelock-free* if for any thread a and any history H in $\mathcal{S}$ there exists an $n > 0$ such that in any H extension $H' \in \mathcal{S}$, if a performs more than n commit operations in H' after H, *some* commit operation will succeed in H' after H (not necessarily one of a's).

Blocking: A sequential specification $\mathcal{S}$ is said to be *nonblocking* if for any thread a and any history H in $\mathcal{S}$ there exists an $n > 0$ such that in any H extension $H' \in \mathcal{S}$, if all operations in H' after H are performed by a, and they include at least n commit operations, at least one of those commits will succeed.

Note that these conditions are defined here at the level of transactions. If extended in the obvious way to concurrent histories of implementations, they yield, respectively, the familiar notions of wait freedom, lock freedom, and obstruction freedom [6, 8].

Lemma 6. For any validity-ensuring conflict function C, C-based TM admits blocking.

Proof: Consider histories of the form $H_k = R\ W_1\ W_2\ \ldots\ W_k$, where R is the 2-operation sequence start(r) read(o, r), performed by some thread a, and W_i is the 3-operation sequence start(w_i) write(o, i, w_i) commit(w_i), performed by some thread b. Since C ensures consistency, transaction R conflicts with all transactions

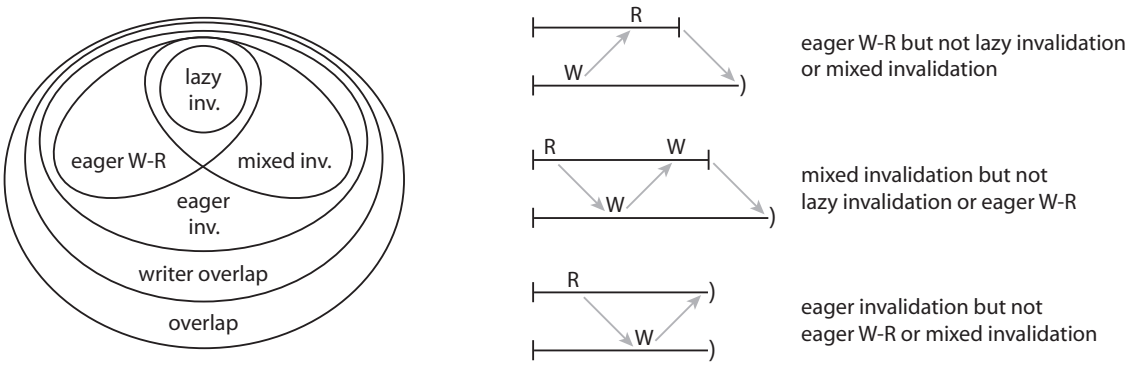

Figure 2. Left: containment relationships among sets of conflicting transactions. Smaller sets provide fewer excuses for a transaction to fail. Right: timelines illustrating histories that separate the inner sets. Arrows indicate history order.

W_i. Thus given any $n > 0$, C-based TM contains a version of H_n in which b performs all operations after R, including n commits, all of which fail. □

Note that an implementation is not *required* to fail all the writes in this example; the point is that C-based TM *permits* it to do so.

Corollary 1. For any validity-ensuring conflict function C, C-based TM admits livelock and starvation.

If we want to ensure progress, clearly we need to insist that some transactions succeed even in the presence of conflicts. To do so, we introduce a function to arbitrate between pairs of conflicting transactions. We can then insist that a transaction succeed if there is no conflicting transaction to which it loses at arbitration.

Where conflict is a purely local phenomenon, based only on the operations of the conflicting transactions, we allow arbitration to consider a broader context. Let $H_{[\![\mathsf{s},\mathsf{t}]\!]}$ be the prefix of H extending through the earlier of $\mathsf{commit}(\mathsf{t})$, $\mathsf{abort}(\mathsf{t})$, $\mathsf{commit}(\mathsf{s})$, or $\mathsf{abort}(\mathsf{s})$ in H. We define an ***arbitration function*** A to be a mapping from $\mathcal{H} \times \mathcal{D} \times \mathcal{D}$ to $\{\mathsf{true}, \mathsf{false}\}$ such that (1) $A(H, \mathsf{s}, \mathsf{t})$ is undefined if $\mathsf{s} = \mathsf{t}$; (2) $\neg A(H, \mathsf{s}, \mathsf{t}) \rightarrow A(H, \mathsf{t}, \mathsf{s})$ if $\mathsf{s} \neq \mathsf{t}$; and (3) if $H_{[\![\mathsf{s},\mathsf{t}]\!]} = I_{[\![\mathsf{s},\mathsf{t}]\!]}$, then $A(H, \mathsf{s}, \mathsf{t}) = A(I, \mathsf{s}, \mathsf{t})$.

If transactions S and T conflict in H and $A(H, S, T) = \mathsf{true}$, transaction S must fail. It seems likely that many arbitration functions will satisfy $\neg A(H, \mathsf{s}, \mathsf{t}) \leftrightarrow A(H, \mathsf{t}, \mathsf{s})$, but our definitions do not require this. A history H is said to be *AC-respecting*, for some conflict function C and arbitration function A, if (1) for every pair of transactions S and T in H, if $C(H, S, T) = \mathsf{true}$, then S fails if $A(H, S, T) = \mathsf{true}$, and T fails if $A(H, T, S) = \mathsf{true}$; and (2) for every transaction T in H, if T ends with a commit operation, then that operation succeeds unless there exists a transaction S in H such that $C(H, T, S) = \mathsf{true}$ and $A(H, T, S) = \mathsf{true}$. *$AC$-based transactional memory* denotes the set of all consistent, AC-respecting histories.

Theorem 4. For any conflict function C and arbitration function A, AC-based TM is a sequential specification.

Proof: Analogous to that of Theorem 2.

As a simple example, we can extend the semantics of overlap-respecting histories with an arbitration function that chooses as victim the transaction that started first:

Eagerly aggressive arbitration: For transactions S and T in history H, $A(H, S, T) = \mathsf{true}$ if $\mathsf{start}_S <_H \mathsf{start}_T$.

A trivial implementation of eagerly aggressive, overlap-based TM might keep the descriptor of the most recently started transaction in a global variable. Operation $\mathsf{start}(\mathsf{t})$ would store t in this variable; $\mathsf{commit}(\mathsf{t})$ would return true iff the variable were still t.

Lemma 7. Eagerly aggressive, overlap-based TM is nonblocking.

Proof: Given any history $H \in$ eagerly aggressive, overlap-based TM and any thread a, consider any extension H' of H composed entirely of operations of a after H. If H' contains two commit operations after H then H' contains a full transaction T of a after H, during which no other transaction starts. By the definition of eagerly aggressive, overlap-based TM, T must be successful. □

Eagerly aggressive, overlap-based TM retains, trivially, the vulnerability to livelock of ordinary overlap-based TM. One way to eliminate this problem is to resolve conflicts in favor of the transaction that attempts to commit first:

Lazily aggressive arbitration: For transactions S and T in history H, $A(H, T, S) = \mathsf{true}$ if $\mathsf{commit}_S <_H \mathsf{end}_T$ and for all transactions U such that $\mathsf{commit}_U <_H \mathsf{commit}_S$, $C(H, U, S) = \mathsf{false}$ or $A(H, U, S) = \mathsf{true}$. That is, T must fail if it conflicts with S, S commits first, and S is not itself forced to fail by some earlier transaction.

Eagerly and lazily aggressive arbitration both resolve conflicts in favor of the thread that "discovers" the conflict. More precisely, in both cases the shortest history prefix in which the value of the arbitration function is defined ends with an operation of the "winning" thread.

Theorem 5. For any conflict function C, lazily aggressive C-based TM is livelock free.

Proof: Suppose the contrary: there exists a history $H \in$ lazily aggressive C-based TM, a thread a, and a prefix P of H such that a performs two commit operations after P in H, neither of which succeeds. Consider the second commit. Call its transaction T. How can T fail? By the definition of lazily aggressive arbitration, there must be some conflicting transaction S in H such that $\mathsf{commit}_S <_H \mathsf{commit}_T$ and S is not forced to fail by any earlier transaction U. Moreover since $C(H, U, S)$ considers only operations prior to the earlier of end_U and end_S, S cannot be forced to fail by any later transaction. By the definition of arbitration, S must succeed. Moreover since T starts after P, S commits after P, contradicting our assumption. □

NB: since sequential specifications say nothing about concurrent histories, it is still possible for a concurrent implementation of a nonblocking, livelock-free specification to have operations that block or livelock.

Theorem 6. For any validity-ensuring conflict function C, lazily aggressive C-based TM admits starvation.

Proof: Consider histories of the form $H_k = W_1\ W_2 \ldots W_k$, where W_i is the 6-operation sequence start(a_i) start(b_i) read(o, a_i) write(o, i, b_i) commit(b_i) commit(a_i), where all the a transactions are performed by the same thread a. Since C ensures consistency, each b transaction conflicts with the corresponding a transaction. And by the definition of lazily aggressive arbitration, the a transaction always loses. Thus given any $n > 0$, lazily aggressive C-based TM contains exactly one version of H_n, in which thread a is never successful. □

Claim (Proof omitted)**.** OSTM is an implementation of lazily aggressive, lazy invalidation-based TM. Even in the absence of adversarial scheduling, it admits the possibility that a thread will starve if it tries, repeatedly, to execute a long, complex transaction in the face of a continual stream of short conflicting transactions in other threads.

Contention management. While it may seem natural for a sequential specification to specify the outcome of conflicts, there are two potentially serious disadvantages to doing so. First, if we attempt to capture some nontrivial notion of fairness in our arbitration function (based, perhaps, on how often the threads in question have lost at arbitration in the past), we may end up with an undesirably complicated specification, or one that over-constrains the implementation (e.g., by requiring guarantees where heuristic or probabilistic assurances might be acceptable in practice). Second, we may preclude decisions based on factors outside the purview of the specification (e.g., thread priorities, processor load, or run-time cache performance.)

An attractive alternative strategy is to couple a blocking or livelock-admitting sequential specification with an implementation that avoids the histories in which blocking or livelock occurs. In effect, this is the suggestion of Herlihy, Luchangco, and Moir [6, 7], who argue for *obstruction-free* algorithms. In such an algorithm the implementation subsumes the role of an arbitration function, which can then be realized as a self-contained *contention management* module. So long as it follows certain minimal rules, a contention manager can guarantee forward progress without the design and verification complexity that would be required for direct implementation of a comparable arbitration function embedded in the specification. A variety of sophisticated contention managers, several of them quite subtle, have been developed in recent years [2, 3, 4, 12, 13, 14].

6. Object-based API

As noted in Section 1, our model of transactional memory is intended to simplify correctness arguments, not to simplify programming. Several extensions are useful in practice, and indeed are embodied in extant TM systems. We focus in this Section on object-oriented software TM systems such as DSTM [7], OSTM [1], ASTM [9], SXM [2], and RSTM [10]. Our extensions are straightforward optimizations and wrappers for the TM operations used in Sections 1 through 5; they do not change the underlying semantics. Simpler extensions, not presented here, would adapt our TM model to hardware TM proposals.

We use each object in the TM model to represent a reference to a higher-level object, and require that (1) the pointer value passed to write is always new (created in the current transaction), and (2) the data to which it refers is never modified after the writing transaction commits or aborts.

To avoid wasting work in a transaction that is doomed to fail, we provide an acquire(o, d, t) operation that does what write does, but returns a Boolean status. If the status is false, the TM has determined (via eager conflict detection) that a subsequent commit is guaranteed to fail. The transaction may then choose to call abort immediately, rather than proceeding. In a similar vein, open(o, t) takes the place of read, and returns nil (distinct from $\perp$) if a subsequent commit is doomed to fail.

To eliminate the prohibition against multiple calls to write in a single transaction, we implement an open_w(o) operation:

```
if open_w has already been called on o in this transaction
    return what it returned last time
else
    d1 := read(o)
    d2 := pointer to new data
          initialized to be a copy of *d1
    if ! acquire(o, d2, t) then d2 := nil
    return d2
```

The intent here is that changes to program data will be made indirectly through the reference returned by open_w. The penultimate line eliminates the need for explicit calls to acquire.

By analogy to open_w, we provide a memoizing open_r(o):

```
if open_r or open_w has already been called on o in this
        transaction
    return what it returned last time
else return read(o)
```

Clearly, calls to open_r always return the same value in the same transaction.

Validation. While Theorem 1 ensures that successful transactions see a sequentially consistent view of memory, it does not ensure that values read from different objects in a failed transaction will be *mutually consistent*—there may be no point in the serialized history at which those values were simultaneously valid. Absent complete sandboxing of transactional operations (implemented via compiler support or binary rewriting), inter-object inconsistency can compromise program correctness in potentially catastrophic ways. In particular, use of an invalid code or data pointer can lead to modification of an arbitrary (nontransactional) data location, or execution of arbitrary code.

We posit a validate(o, d) operation, implemented as return (read(o) = d), that can be used to verify that a value is still valid. DSTM, ASTM, and RSTM ensure consistency automatically and incrementally, by having open_r and open_w call validate for every previously-opened object. OSTM requires the programmer to insert such calls by hand whenever the use of inconsistent data might lead to unacceptable behavior.

7. Conclusions

In this note we have suggested that transactional memory be viewed not merely as a means of implementing concurrent objects, but as a concurrent object in its own right. Toward that end we considered the sequential specification of transactional memory semantics. We suggested that any intuitively acceptable specification of TM consist of all and only those histories in which all read operations of successful transactions return the "right" value, and no commit operation fails unless provided an excuse to do so by some well-defined *conflict function*, optionally augmented with an *arbitration function*. We presented a collection of conflict functions that overlap in nontrivial ways, inducing a rich collection of sequential specifications. We noted that deferring the work of an arbitration function to the implementation corresponds to the notion of *contention management* in obstruction-free STM.

Several of our sequential specifications capture the semantics of published TM systems. The formalization exercise also leads us to suggest that *mixed invalidation*-based TM (eager detection of write-write conflicts, lazy detection of read-write conflicts) might be an option worth exploring in future TM systems. Regarding the

formalization itself, our work suggests a variety of open questions, among them:

- Should we extend the notion of consistency to allow a read in a successful transaction to return a stale or, conversely, a not-yet-committed value?
- Can we characterize the circumstances under which a read in a failed or aborted transaction is permitted to return an "incorrect" value?
- How sophisticated an arbitration function can realistically be embedded in a sequential specification? Are there any advantages to including it there, rather than leaving it to the implementation?
- Can we characterize the conflict and arbitration functions that do or do not lead to blocking or livelock-admitting specifications?
- Can we develop a meaningful notion of probabilistic arbitration functions?
- Can we create an arbitration function that precludes starvation, or would this require extensions to the model of Section 1 (e.g., to allow the specification of continuations)?
- Is there any potential benefit to extending the definition of conflict function to allow two non-overlapping transactions to conflict? This might, among other things, allow certain isolated transactions to fail.
- Is there any call for a weaker notion of "validity-ensuring conflict function" that would exploit value-restoring (ABA) writes?

Acknowledgments

The ideas in this paper benefited greatly from the comments of the anonymous referees, and from discussions with Bill Scherer, David Eisenstat, Virendra Marathe, Mike Spear, and Mitsu Ogihara.

References

[1] K. Fraser and T. Harris. Concurrent Programming Without Locks. Submitted for publication, 2004. Available as research.microsoft.com/~tharris/drafts/cpwl-submission.pdf.

[2] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management in SXM. In *Proceedings of the Nineteenth International Symposium on Distributed Computing*, Cracow, Poland, September 2005.

[3] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust Contention Management in Software Transactional Memory. In *Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, October 2005. In conjunction with OOPSLA'05.

[4] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a Theory of Transactional Contention Managers. In *Proceedings of the Twenty-Fourth ACM Symposium on Principles of Distributed Computing*, Las Vegas, Nevada, August 2005.

[5] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *OOPSLA 2003 Conference Proceedings*, Anaheim, CA, October 2003.

[6] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of the Twenty-Third International Conference on Distributed Computing Systems*, Providence, RI, May, 2003.

[7] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing*, pages 92–101, Boston, MA, July 2003.

[8] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[9] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *Proceedings of the Nineteenth International Symposium on Distributed Computing*, Cracow, Poland, September 2005.

[10] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Software Transactional Memory. In *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, ON, Canada, July 2006. Held in conjunction with PLDI 2006. Expanded version available as TR 893, Department of Computer Science, University of Rochester, March 2006.

[11] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Third edition, 2003.

[12] W. N. Scherer III and M. L. Scott. Contention Management in Dynamic Software Transactional Memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, July 2004.

[13] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the Twenty-Fourth ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[14] W. N. Scherer III and M. L. Scott. Randomization in STM Contention Management (poster paper). In *Proceedings of the Twenty-Fourth ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.