Memory Models for Open-Nested Transactions

Kunal Agrawal Charles E. Leiserson Jim Sukha
MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139, USA

ABSTRACT

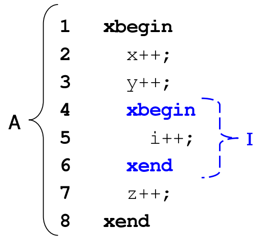
Open nesting provides a loophole in the strict model of atomic transactions. Moss and Hosking suggested adapting open nesting for transactional memory, and Moss and a group at Stanford have proposed hardware schemes to support open nesting. Since these researchers have described their schemes using only operational definitions, however, the semantics of these systems have not been specified in an implementation-independent way. This paper offers a framework for defining and exploring the memory semantics of open nesting in a transactional-memory setting.

Our framework allows us to define the traditional model of ***serializability*** and two new transactional-memory models, ***race freedom*** and ***prefix race freedom***. The weakest of these memory models, prefix race freedom, closely resembles the Stanford open-nesting model. We prove that these three memory models are equivalent for transactional-memory systems that support only closed nesting, as long as aborted transactions are "ignored." We prove that for systems that support open nesting, however, the models of serializability, race freedom, and prefix race freedom are distinct. We show that the Stanford TM system implements a model at least as strong as prefix race freedom and strictly weaker than race freedom. Thus, their model compromises serializability, the property traditionally used to reason about the correctness of transactions.

1. INTRODUCTION

Atomic transactions represent a well-known and useful abstraction for programmers writing parallel code. Database systems have utilized transactions for decades [9], and more recently, transactional memory [12] has become an active area of research. Transactional memory (TM) describes a collection of hardware and software mechanisms that provide a transactional interface for accessing memory, as opposed to a database. A TM system guarantees that any section of code that the programmer has specified as a transaction either appears to execute atomically or appears not to happen at all, even though other transactions may be running concurrently. In the first case, we say the transaction has ***committed***; otherwise, we say the transaction has ***aborted***.

A TM system enforces atomicity by tracking the memory locations that each transaction in the system accesses, finding transaction conflicts, and aborting and possibly retrying transactions to resolve conflicts. Most TM implementations maintain a transaction ***readset*** and ***writeset***, i.e., a list of memory locations that a transaction has read from or written to, respectively. Typically, the system reports a conflict between two transactions A and B if both transactions access the same memory location and at least one of those accesses is a write. If A and B conflict, then TM aborts one of the transactions, rolls back any changes the aborted transaction made to global memory, and clears its readset and writeset.



```
xbegin
  x++;
  y++;
  xbegin
    i++;
  xend
  z++;
xend
```

FIGURE 1: A code example where transaction I is nested inside A. The `xbegin` and `xend` delimiters mark the beginning and end of a transaction.

Transactional memory systems may support ***nesting*** of transactions. Nested transactions arise when an outer transaction A in its body calls another transaction I. Figure 1 shows code for a transaction A within which another transaction I is nested.

The database community has produced an extensive literature on nested transactions. Moss [17] credits Davies [4] with inventing nested transactions, and he credits Reed [23] as providing the first implementation of what we now call closed transactions. Gray [8] describes what we now call open transactions. The terms "open" and "closed" nesting" were coined by Traiger [25] in 1983.

The TM literature discusses three types of nesting: flat, closed, and open. The semantics and performance implications of each form of nesting can be understood through the example of Figure 1. If I is ***flat-nested*** inside A, then conceptually, A executes as if the code for I were inlined inside A. With flat-nesting, I's reads and writes are added directly to the readset and writeset of A. Thus, in Figure 1, if a concurrent transaction B tries to modify variable `i` while I is running, but before I has committed, then if I aborts, it also causes A to abort (since `i` belongs to the readset of A as well).

If I is ***closed-nested*** inside A (see, for example, [18]), then conceptually, the operations of I only become part of A when I commits. In Figure 1, if B tries to modify `i` and causes I to abort, then the system only needs to abort and roll back I, but B need not abort A, because A has not accessed location `i` yet. Thus, closed nesting can be more efficient than flat nesting in this example. I's readset and writeset are merged with A's readset and writeset if I commits, however. Thus, if B tries to modify `i` after I has committed but before A commits, the system may still abort A.

Finally, if I is ***open-nested*** inside A (see [16, 19, 21, 26]), then conceptually, the operations of I are not considered as part of A. When I commits, I's changes are made visible to any other transaction B immediately, in the scheme of [16],[1] independent of whether

This research was supported in part by NSF Grants CNS-0615215 and CNS-0540248, and a gift from Intel Corporation.

MSPC'06 October 22, 2006, San Jose, CA, USA
Copyright © 2006 ACM 1-59593-578-9/06/0010. . . $5.00.

[1] Several alternative policies for manipulating readsets and writesets are suggested in both [19,21], but since [19] suggests adopting the same scheme as [16], we do not discuss the alternatives in this paper.

A later commits or aborts. Thus, in Figure 1, B never aborts A, and B's access to variable `i` is never added to A's readset or writeset.

Transactional memory with either flat or closed nesting guarantees that transactions are ***serializable*** [22]: they affect global memory as if they were executed one at a time in some order, even if in reality, several executed concurrently. Closed nesting generally allows for a more efficient implementation compared with flat nesting, because closed nesting allows a nested transaction I to abort without forcibly aborting its parent transaction A, as with flat nesting.

Open nesting provides a loophole in the strict guarantee of transaction serializability by allowing an outer transaction to "ignore" the operations of its open subtransactions. Moss [19] describes open nesting as a high-level construct that operates at two levels of abstraction. Thus, open nesting may require high-level constructs for rollbacks of aborted transactions or for concurrency control between transactions. For example, when using open nesting, programmers may need to specify a "compensating" transaction that undoes the effect of a committed open transaction if its parent transaction aborts, or the programmer may need to use "abstract" locks in the code to prevent certain transaction interleavings [19].

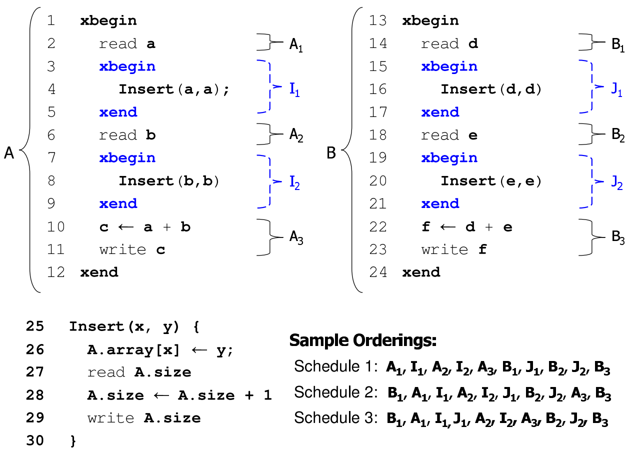
Indeed, even TM without any nesting can be viewed at two levels of abstraction. For example, the hardware may implement rollback of memory state, but rely on the programmer or compiler to retry transactions that abort, sometimes using backoff protocols to ensure that a given transaction eventually commits. Thus, it is helpful to distinguish the *memory model* for TM, as the essential memory semantics that the hardware implements, from the *program model*, as the semantics that the programmer sees.

Our focus will be on memory models for TM. We shall not concern ourselves with retry mechanisms, compensating transactions, and the like. A TM system should have well-specified behavior even as a target for compilation, when all program-level support for transactions and nesting are put aside. Low-level software may build upon the memory model to provide a higher level of abstraction, e.g., for open nesting, but the semantics of open nesting must be understood by the programmers of this low-level software.

Moreover, although one may ignore the semantics of aborted transactions at the program-model level, at the level of the memory model, even aborted transactions must have a reasonable semantics, at least up to the point where they abort. Thus, we shall be interested in defining memory semantics even for aborted transactions.

In this paper, we describe a framework for defining transactional memory models. Our framework, which is inspired by the computation-centric framework proposed by Frigo [6, 7], allows TM semantics to be specified in an implementation-independent way. Within this framework, we define the traditional model of serializability and two new transactional memory models, race freedom and prefix-race freedom. We prove that these three memory models are equivalent for computations that contain only closed transactions, as long as aborted transactions are "ignored." For systems that support open nesting, however, the three models are distinct. We show that the Stanford system [16], perhaps the most reasonable design for open-nesting of transactional memory proposed to date, implements a model at least as strong as prefix-race freedom and strictly weaker than race freedom. Thus, their model compromises serializability, the property traditionally used to reason about the correctness of transactions.

The remainder of this paper is organized as follows. Section 2 presents several examples that illustrate program behaviors that open nesting can admit. Section 3 defines our framework for understanding transactional memory models. Section 4 formally defines the memory models of serializability, race freedom, and prefix-race freedom. In Section 5 we prove that all three memory models are equivalent for computations with only committed transactions, but are distinct when we model aborted transactions or have open transactions. Section 6 describes an operational model for open nesting that similar to the Stanford model [16] and shows that it implements prefix-race freedom. Section 7 offers some perspective on open nesting and other loopholes in transactional memory.



```
xbegin
  read a
  xbegin
    Insert(a,a);
  xend
  read b
  xbegin
    Insert(b,b)
  xend
  c ← a + b
  write c
xend
```

```
xbegin
  read d
  xbegin
    Insert(d,d)
  xend
  read e
  xbegin
    Insert(e,e)
  xend
  f ← d + e
  write f
xend
```

```
Insert(x, y) {
  A.array[x] ← y;
  read A.size
  A.size ← A.size + 1
  write A.size
}
```

Sample Orderings:

Schedule 1: $A_1, I_1, A_2, I_2, A_3, B_1, J_1, B_2, J_2, B_3$

Schedule 2: $B_1, A_1, I_1, A_2, I_2, J_1, B_2, J_2, A_3, B_3$

Schedule 3: $B_1, A_1, I_1, J_1, A_2, I_2, A_3, B_2, J_2, B_3$

FIGURE 2: Two concurrent transactions that do not share any memory locations except in their nested transactions. Divide transaction A into abstract operations A_1, I_1, A_2, I_2, A_3, and divide B into B_1, J_1, B_2, J_2, B_3. The I's and J's represent inserts to an abstract table data structure. Schedule 1 is a serial order, Schedule 2 is an interleaved order equivalent to Schedule 1, and Schedule 3 is an interleaved order which is not serializable.

2. SUBTLETIES WITH OPEN NESTING

This section motivates the need for a precise description of the memory semantics using three examples to illustrate some subtleties with open nesting. The first example shows that some desirable schedules allowed by open nesting are not serializable. The second example shows that the loss of serializability for open nesting sanctions arguably bizarre program behaviors. The third example shows that open nesting compromises composability.

Figure 2 describes a program with nested transactions where the use of open nesting admits a desirable schedule which is not serializable. Moreover, a system with only flat or closed nesting prohibits the schedule. In Figure 2, transaction A reads from global variable a, adds a key-value pair based on a to a global table, reads from b and adds a corresponding pair to the table, and then stores the sum $a + b$ into c. Transaction B performs analogous operations on d, e, and f. The table data structure is implemented as a simple direct-access table [3, Section 11.1] with a global `size` field to count the number of elements in the table.

If the nested transactions (the I's and J's) are all flat-nested or closed-nested, then TM guarantees that the transactions are serializable: the program appears to executes as though either A happened before B (Schedule 1) or B happened before A. The system might actually perform the operations in a different, interleaved order (for example, Schedule 2), but this schedule is equivalent to one of the two valid serial schedules (in this case, Schedule 1). Schedule 3 is not serializable, however, because J_1 (and thus B) observes the intermediate value of `A.size` written by I_1 (and thus written by A). Consequently, Schedule 3 is prohibited with flat or closed nesting.

To improve concurrency, a programmer may wish to allow certain schedules that are not serializable, but which nevertheless are consistent from the programmer's point of view. A system that can admit nonserializable schedules imposes fewer restrictions on transactions, possibly allowing transactions to commit when they would have otherwise aborted. For example, the programmer may

```
    ┌ 1  xbegin
    │ 2    read a              ]- A1
    │ 3    xbegin_open         ┐
    │ 4      read i            │
    │ 5      i ← i + 1         ├- I1
    │ 6      write i           │
A  <  7    xend_open           ┘
    │
    │ 18   read b              ┐
    │ 19   c ← a + b           ├- A2
    │ 20   write c             ┘
    └ 21 xend

    ┌ 8  xbegin
    │ 9    read i
B  <  10   i ← i + 1
    │ 11   write i
    └ 12 xend

    ┌ 13 xbegin
    │ 14   read i
C  <  15   b ← i
    │ 16   write b
    └ 17 xend
```

FIGURE 3: A program execution permitted by open nesting. Transaction A does not appear to execute atomically, because it can read an "inconsistent" value for b if B and C interleave between the execution of A_1 and A_2.

wish to admit Schedule3, even though the I's and J's happen to access the same size field. Conceptually, the programmer may not care in which order the table inserts occur. For example, if I_1, I_2, J_1, and J_2 are open transactions, then Schedule 3 is a valid execution.

Once a TM system with open nesting admits some desirable nonserializable schedules, however, the proverbial cat is out of the bag. As far as the memory semantics are concerned, it seems difficult to prohibit additional program behaviors that might arguably be undesirable. Figure 3 shows a program execution allowed by the open-nesting implementations of [16,21]. In this example, it is possible for all transactions A, I_1, B, and C to commit, even though A does not appear to execute atomically. Transaction A reads inconsistent data, since C writes to b between A's reads of a and b. Thus, the "snapshot" of the world seen by A when it begins is different from its snapshot part way through its computation.

Our final example illustrates how open nesting can admit subtle program behaviors that affect the composability of transactions. Consider the program in Figure 4 which describes an implementation of a simple table library that (arguably) contains an subtle flaw. The program includes a `Contains(x)` method to complement the `Insert(x,y)` method used in Figure 2. Since the `size` field is the primary source of transaction conflicts between table operations, the `Contains` method "optimizes" its search method by checking `size` within an open transaction.

Using TM with open nesting, in any sequence of `Contains` or `Insert` operations, each individual operation still appears atomic. Thus, in transaction A in Figure 4, we might expect that if the `Contains` operation returns false, then the key can be safely inserted into the hash table without adding duplicates.

Unfortunately, one cannot correctly call both `Contains` and `Insert` inside a transaction T and still have T appear to be atomic. Indeed, the open-nesting implementation described in [16] allows the entire transaction B to execute between Lines 2 and 7 of transaction A. Thus, this code shows that composability of transactions is not preserved. When using open nesting, simply ensuring the atomicity of individual transactions is not sufficient to guarantee composability.

Admittedly, the examples in Figures 3 and 4 are somewhat contrived. In particular, unlike in Figure 2, transactions in Figures 3 and 4 cannot be partitioned into clear abstraction levels, with each level accessing disjoint memory locations, as Moss suggests may be necessary [19]. These examples suggest, however, that for open nesting, the distinction between the abstract program model and the low-level memory model is much more significant than for closed or flat nesting. Thus, these examples motivate the need to understand memory models for open nesting so that at the very least we can understand what properties should be enforced by higher-level mechanisms.

```
   ┌ 1  xbegin                          ┌ 3  xbegin
A <  2    if (!Contains(5))          B <  4    if (!Contains(5))
   │ 7      Insert(5,15)                │ 5      Insert(5,10)
   └ 8  xend                            └ 6  xend

Contains(x)                          Insert(x,y)
9   xbegin                           19  xbegin
10    found ← true                   20    A.array[x] ← y
11    xbegin_open                    21    read A.size
12      read A.size                  22    A.size ← A.size + 1
13      if (A.size==0) empty← true   23    write A.size
14    xend_open                      24  xend
15    if (!empty)
16      found ← A.array[x]
17    return ((!empty) && found)
18  xend
```

FIGURE 4: Flawed implementation of a table data structure with two methods, `Contains(x)` and `Insert(x,y)`. Although each method individually appears atomic, transactions A and B, which call those methods, may not appear atomic. In particular, the ordering $\langle 1, 2, 3, 4, 5, 6, 7, 8\rangle$ is allowed.

3. MEMORY MODELS

This section defines our framework for modeling transactional computations. Our model is inspired by Frigo's computation-centric modeling of a program execution as a computation dag (directed acyclic graph) [6] with an "observer function" which essentially tells what write operation is "seen" by a read. Our model uses a "computation tree" to model both the computation dag and the nesting structure of transactions. We first define computation trees without transactions, then we show how transactions can be specified, and finally, we define Lamport's classical sequential-consistency model [14].

Formal models for systems with nested transactions appear as early as the work by Beeri, Bernstein, and Goodman [1]. Recent papers providing operational semantics for open transactions include [15, 16, 21]. Although operational semantics of a TM can provide an abstract basis for implementation, inferring emergent properties of the system from these semantics can be quite difficult.

Our computation-centric model focuses on an *a posteriori* analysis of a program execution. After a program completes, we assume the execution has generated a ***trace*** which is abstractly modeled as a pair (C, Φ), where C is a "computation tree" describing the memory operations performed and transactions executed, and Φ is an "observer function" describing the behavior of read and write operations. We shall define C and Φ more precisely below. We define $\mathcal{U}$ to be the set of all possible traces (C, Φ).

Within this framework, we define a memory model as follows:

DEFINITION 1. *A **memory model** is a subset $\Delta \subseteq \mathcal{U}$.*

That is, Δ represents all executions that "obey" the memory model.

Computation trees without transactions

The computation tree C summarizes the information about the control structure of a program together with the structure of nested transactions. We first describe how a computation tree models the structure of a program execution in the special case where the computation has no transactions.

Structurally, a ***computation tree*** C is an ordered tree with two types of nodes: ***memory-operation nodes*** $\texttt{memOps}(C)$ at the leaves, and ***control nodes*** $\texttt{spNodes}(C)$ as internal nodes. Let $\texttt{nodes}(C) = \texttt{memOps}(C) \cup \texttt{spNodes}(C)$ denote the set of all nodes of C.

We define $\mathcal{M}$ to be the set of all memory locations. Each leaf node $u \in \texttt{memOps}(C)$ represents a single memory operation on a memory location $\ell \in \mathcal{M}$. We say that node u satisfies the ***read predicate*** $R(u, \ell)$ if u reads from location ℓ. Similarly, u satisfies the ***write predicate*** $W(u, \ell)$ if u writes to ℓ.

The internal nodes $\texttt{spNodes}(C)$ of C represent the parallel control structure of the computation. In the manner of [5], each internal node $X \in \texttt{spNodes}(C)$ is labeled as either an S-node or P-node to capture fork/join parallelism. All the children of an S-node are executed in series from left to right, while the children of an P-node can be executed in parallel.

Several structural notations will help. Denote the ***root*** of a computation tree C as $\texttt{root}(C)$. For any internal node $X \in \texttt{spNodes}(C)$, let $\texttt{children}(X)$ denote the ordered set of X's children. For any tree node $X \in \texttt{nodes}(C)$, let $\texttt{ances}(X)$ denote the set of all ancestors of X in C, and let $\texttt{desc}(X)$ denote the set of all X's descendants. Denote the set of proper ancestors (and descendants) of X by $\texttt{pAnces}(X)$ (and $\texttt{pDesc}(X)$). Denote the ***least common ancestor*** of two nodes $X_1, X_2 \in C$ by $\texttt{LCA}(X_1, X_2)$.

Since every subtree of a computation tree is also a computation tree, we shall sometimes overload notation and use a subtree and its root interchangeably. For example, if $X = \texttt{root}(C)$, then $\texttt{memOps}(X)$ refers to all the leaf nodes in C, and $\texttt{children}(C)$ refers to the children of X.

Computation dags

A computation tree C defines a ***computation dag*** $G(C) = (V(C), E(C))$ constructed as follows and illustrated in Figure 5. For every internal node $X \in \texttt{spNodes}(C)$, we create and place two corresponding vertices, $\texttt{begin}(X)$ and $\texttt{end}(X)$ in $V(C)$. For every leaf node $x \in \texttt{memOps}(C)$, we place the single node x in $V(C)$. For convenience, for all $x \in \texttt{memOps}(C)$, we define $\texttt{begin}(x) = \texttt{end}(x) = x$.

Formally, the vertices of the graph $V(C)$ are defined as follows:

$$V(C) = \texttt{memOps}(C) \cup \left(\bigcup_{X \in \texttt{spNodes}(C)} \{\texttt{begin}(X), \texttt{end}(X)\} \right) .$$

For any computation tree rooted at node X, we define the edges $E(X)$ for the graph $G(X)$ recursively:
Base case: If $X \in \texttt{memOps}(C)$, then define $E(X) = \emptyset$.
Inductive case: If $X \in \texttt{spNodes}(C)$, let $\texttt{children}(X) = \{Y_1, Y_2, \ldots, Y_k\}$. If X is an S-node, then

$$\begin{aligned} E(X) = &\{(\texttt{begin}(X), \texttt{begin}(Y_1)), (\texttt{end}(Y_k), \texttt{end}(X))\} \\ &\cup \left(\bigcup_{i=1}^{k-1} \{(\texttt{end}(Y_i), \texttt{begin}(Y_{i+1}))\} \right) \cup \left(\bigcup_{i=1}^{k} E(Y_i) \right) . \end{aligned}$$

If X is a P-node, then

$$\begin{aligned} E(X) = &\left(\bigcup_{i=1}^{k} E(Y_i) \right) \\ &\cup \left(\bigcup_{i=1}^{k} \{(\texttt{begin}(X), \texttt{begin}(Y_i)), (\texttt{end}(Y_i), \texttt{end}(X))\} \right) . \end{aligned}$$

We shall find it convenient to overload the $\texttt{LCA}$ function, and define the least common ancestor of two graph vertices $u, v \in V(C)$ as the $\texttt{LCA}$ of the corresponding tree nodes.

The computation dag $G(C)$ is a convenient way of representing the flow of the program execution specified by C. Unfortunately, our specification of computation dags via computation trees limits the set of computation dags that can be described. In particular, computation trees can only specify "series-parallel" dags [5]. We might have founded our framework for transactional-memory semantics on more-general computational dags, but the added generality would not affect any of our theorems, and it would have greatly complicated definitions and proofs.

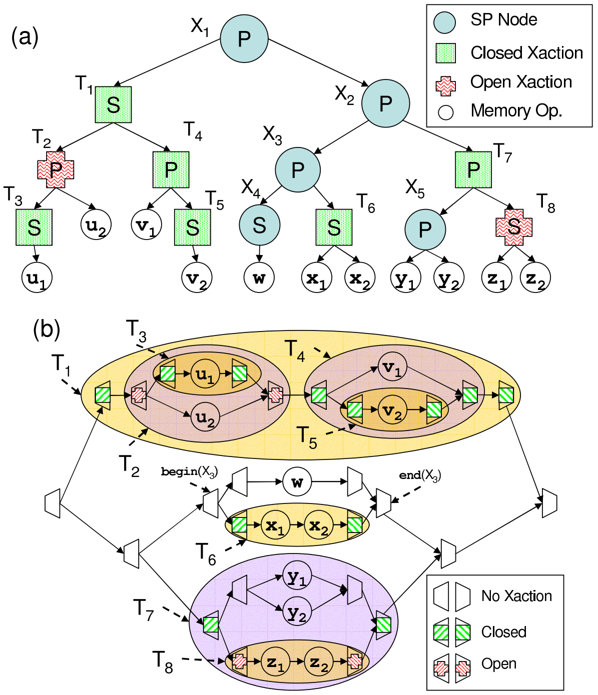


FIGURE 5: A sample (a) computation tree C and (b) the corresponding dag $G(C)$ for a computation that has closed and open transactions. In this example, T_2 is open-nested inside T_1 and T_8 is open-nested inside T_7. The X_i's are tree nodes that are not marked as transactions. We have not specified whether each transaction is committed or aborted.

We shall find it useful to define some graph notations. For a graph $G = (V, E)$ and vertices $u, v \in V$, we write $u \preceq_G v$ if there exists a path from u to v in G, and we write $u \prec_G v$ if $u \neq v$ and $u \preceq_G v$. For any dag $G = (V, E)$, a ***topological sort*** $\mathcal{S}$ of G is an ordering of all the vertices of V such that for all $u, v \in V$, we have $u \prec_G v$ implies that $u <_{\mathcal{S}} v$ (u comes before v in $\mathcal{S}$). For a dag G, we define $\texttt{topo}(G)$ as the set of all topological sorts of G.

Transactional computation trees

We can specify transactions in a computation tree C by marking internal tree nodes. Marking a node $T \in \texttt{spNodes}(C)$ as a transaction corresponds to defining a transaction T that contains the computation subdag $G(T)$, where $\texttt{begin}(T)$ is the start of the transaction and $\texttt{end}(T)$ is the end of the transaction.[2] Formally, the computation tree C specifies a set $\texttt{xactions}(C) \subseteq \texttt{spNodes}(C)$ of internal nodes as ***transactions***, and a set $\texttt{open}(C) \subseteq \texttt{xactions}(C)$ of ***open*** transactions. The set of ***closed*** transactions is $\texttt{closed}(C) = \texttt{xactions}(C) - \texttt{open}(C)$. In Figure 5, nodes T_1 through T_8 are transactions, and X_1 through X_5 are ordinary nodes. Define a transaction $T \in \texttt{xactions}(C)$ as ***nested*** inside another transaction $T' \in \texttt{xactions}(C)$ if $T' \in \texttt{ances}(T)$. Two transactions T and T' are ***independent*** if neither is nested in the other.

The computation tree C also specifies a set $\texttt{committed}(C) \subseteq \texttt{xactions}(C)$ of ***committed*** transactions. Similarly, transactions belonging to $\texttt{aborted}(X) = \texttt{xactions}(X) - \texttt{committed}(X)$ are ***aborted*** transactions. For a transaction $T \in \texttt{xactions}(C)$, the ***content*** of T is the set of all operations that belong to T but not to

[2] We assume that every leaf $x \in \texttt{memOps}(C)$ is its own committed, closed transaction, but we do not mark leaves as a transactions in our model.

any of T's open or aborted subtransactions. Formally, we define

$$\texttt{content}(T) = V(T) \; - \bigcup_{Z \in \texttt{open}(T) - \{T\}} V(Z) \; - \bigcup_{Z \in \texttt{aborted}(T) - \{T\}} V(Z) \; .$$

We always have $\texttt{content}(T) \subseteq V(T)$, and equality holds when T's subtree contains no open or aborted transactions.[3] For example, in Figure 5, memory operations u_1 and u_2 do not belong to $\texttt{content}(T_1)$, because T_2 is an open transaction nested within T_1. As another example from the figure, we have $v_2 \in \texttt{content}(T_4)$ if and only if $T_5 \in \texttt{committed}(C)$. We also define the ***holders*** of a vertex $v \in V(C)$ to be the set

$$\texttt{h}(v) = \{T \in \texttt{xactions}(C) : u \in \texttt{content}(T)\}$$

of all transactions that contain v.

Hidden vertices

Basic transactional semantics dictate that committed transactions should not "see" values written by vertices belonging to the content of an aborted transaction. One may argue whether one aborted transaction should be able to see values written by a another aborted transaction. In this paper, we take the position that up to the point that a transaction aborts, it should be "well behaved" and act as if it would commit. The well-behavedness of aborted transactions is implicitly assumed by the various proposals for open nesting [16, 19, 21]. Thus, one aborted transaction should not see values written by other aborted transactions, although the values written by a vertex within an aborted transaction may be seen by other vertices within the same transaction.

The following definition describes which vertices are hidden from which other vertices.

DEFINITION 2. *For any two vertices $u, v \in V(C)$, let $X = \texttt{LCA}(u, v)$. We say that u is **hidden** from v, denoted uHv, if*

$$u \in \bigcup_{Y \in \texttt{aborted}(X) - \{X\}} \texttt{content}(Y) \; .$$

In Figure 5, we have $v_2 H z_2$ if and only if at least one of T_1, T_4, or T_5 belongs to $\texttt{aborted}(C)$. Since T_2 is an open transaction, however, we never have $u_1 H z_2$ if $T_2, T_3 \in \texttt{committed}(C)$, even if $T_1 \in \texttt{aborted}(C)$. If we have $T_1, T_4 \in \texttt{committed}(C)$ and $T_7 \in \texttt{aborted}(C)$, then we also have $y_1 H v_1$, but not $v_1 H y_1$, and thus the hidden relation H is not symmetric.

Observer functions

Instead of specifying the value that a vertex $v \in \texttt{memOps}(C)$ reads from or writes to a memory location $\ell \in \mathcal{M}$, we follow Frigo's computation-centric framework [6, 7] which abstracts away the values entirely. An ***observer function***[4] $\Phi(v) : \texttt{memOps}(C) \rightarrow \texttt{memOps}(C) \cup \{\texttt{begin}(C)\}$ tells us which vertex $u \in \texttt{memOps}(C)$ writes the value of ℓ that v sees. For a given computation tree C, if $v \in \texttt{memOps}(C)$ accesses location $\ell \in \mathcal{M}$, then a well-formed observer function must satisfy $\neg(v \prec_{G(C)} \Phi(v))$ and $W(\Phi(v), \ell)$. In other words, v can not observe a value from a vertex that comes after v in the computation dag, and v can only observe a vertex if it actually writes to location ℓ. To define Φ on all vertices that access memory locations, we assume that the vertex $\texttt{begin}(C)$ writes initial values to all of memory.

Together, a computation tree C and an observer function Φ defined on $\texttt{memOps}(C)$ specify a trace.

Sequential consistency without transactions

We now turn to using our framework to define Lamport's classic model of sequential consistency [14] in our transactional model. We first mimic Frigo's definition [6] to define a sequential-consistency memory model for computations without transactions. We then extend the definition to include transactions as well.

Definition 1 states that a memory model Δ is a subset of $\mathcal{U}$, the universe of all possible traces. Sometimes, we wish to restrict our attention to computations with only closed and/or committed transactions. Thus, we define the following subsets of $\mathcal{U}$:

$$\begin{aligned}
\mathcal{U}_0 &= \{(C, \Phi) \in \mathcal{U} : \texttt{xactions}(C) = \emptyset\} \; , \\
\mathcal{U}_{\text{clo}} &= \{(C, \Phi) \in \mathcal{U} : \texttt{open}(C) = \emptyset\} \; , \\
\mathcal{U}_{\text{com}} &= \{(C, \Phi) \in \mathcal{U} : \texttt{aborted}(C) = \emptyset\} \; , \\
\mathcal{U}_{\text{c\&c}} &= \mathcal{U}_{\text{clo}} \cap \mathcal{U}_{\text{com}} \; .
\end{aligned}$$

In other words, $\mathcal{U}_0$ contains traces (whose computations) include no transactions, $\mathcal{U}_{\text{clo}}$ contains traces that include only closed transactions, $\mathcal{U}_{\text{com}}$ contains traces that include only committed transactions, and $\mathcal{U}_{\text{c\&c}}$ contains traces that include only committed and closed transactions.

We now follow Frigo [6] in defining a "last-writer" observer function.

DEFINITION 3. *Consider a trace $(C, \Phi) \in \mathcal{U}_0$ and a topological sort $\mathcal{S} \in \texttt{topo}(G(C))$. For all $v \in \texttt{memOps}(C)$ such that $R(v, \ell) \vee W(v, \ell)$, the **last writer** of v according to $\mathcal{S}$, denoted $\mathcal{L}_{\mathcal{S}}(v)$, is the unique $u \in \texttt{memOps}(C) \cup \{\texttt{begin}(C)\}$ that satisfies three conditions:*

1. *$W(u, \ell)$,*
2. *$u <_{\mathcal{S}} v$, and*
3. *$\neg \exists w$ s.t $W(w, \ell) \wedge (u <_{\mathcal{S}} w <_{\mathcal{S}} v)$.*

In other words, if vertex v accesses (reads or writes) location ℓ, the last writer of v is the last vertex u before v in the order $\mathcal{S}$ that writes to location ℓ.

We can use the last-writer function to define sequential consistency for computations containing no transactions.

DEFINITION 4. ***Sequential consistency** for computations without transactions is the memory model*

$$SC = \{(C, \Phi) \in \mathcal{U}_0 : \exists \mathcal{S} \in \texttt{topo}(G(C)) \text{ s.t. } \Phi = \mathcal{L}_{\mathcal{S}}\} \; .$$

By this definition, a trace $(C, \Phi) \in \mathcal{U}_0$ is sequentially consistent if there exists a topological sort $\mathcal{S}$ of $G(C)$ such that the observer function Φ satisfies $\Phi(v) = \mathcal{L}_{\mathcal{S}}(v)$ for all memory operations $v \in \texttt{memOps}(C)$. Definition 4 captures Lamport's notion [14] of sequential consistency: there exists a single order on all operations that explains the execution of program. Figure 6 shows a sample computation dag $G(C)$ and two possible observer functions, Φ_1 and Φ_2. The trace (C, Φ_1) is sequentially consistent, but (C, Φ_2) is not.

Transactional sequential consistency

We now extend the definition of sequential consistency to account for transactions. Our definition does not attempt to model atomicity, however — that is the topic of Section 4. It simply models that a transaction outside an aborted transaction cannot "see" values written by the aborted transaction. Moreover, our definition makes the assumption that an aborted computation is consistent up to the point that it aborts.

[3] In this paper, we consider only ***global open nesting***, meaning that if T' is open-nested in T, then it is open with respect to every transaction in $\texttt{ances}(T)$. Alternatively, one might specify T' as ***open-nested with respect to*** an ancestor transaction T. In this case, the operations of T' are excluded from all transactions T'' on the path from T' up to and including T, but included in transactions that are proper ancestors of T. Intuitively, if T' is open-nested with respect to T, then T' commits its changes to T's context rather than directly to memory. Global open-nesting is then the special case when all open transactions are open with respect to $\texttt{root}(C)$.

[4] Our definition of Φ is similar to Frigo's [6], but with a salient difference, namely, Frigo's observer function gives values for all memory locations, not just for the location that a vertex accesses. Moreover, if $W(v, \ell)$, Frigo defines $\Phi(v) = v$, whereas we define $\Phi(v) = u$ for some $u \neq v$.

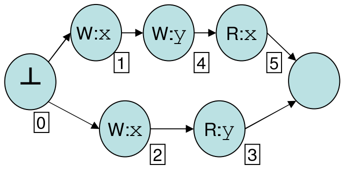


FIGURE 6: Examples of sequential consistency for a computation C with only committed transactions. Shown is the computation dag $G(C)$. For the observer function Φ_1 given by $\langle\Phi_1(1)=0,\Phi_1(2)=1,\Phi_1(3)=0,\Phi_1(4)=0,\Phi_1(5)=2\rangle$, the trace (C,Φ_1) is sequentially consistent, with the topological sort $\mathcal{S}=\langle 0,1,2,3,4,5\rangle$ of $G(C)$. For the observer function Φ_2 given by $\langle\Phi_2(1)=0,\Phi_2(2)=1,\Phi_2(3)=0,\Phi_2(4)=0,\Phi_2(5)=1\rangle$, however, the trace (C,Φ_2) is not sequentially consistent, because there is no topological sort consistent with the last-writer function.

We first redefine the last-writer function to take aborted transactions into account. Intuitively, another transaction should not be able to "see" the values of an aborted transaction.

DEFINITION 5. *Consider a trace $(C,\Phi)\in\mathcal{U}$ and a topological sort $\mathcal{S}\in\texttt{topo}(G(C))$. For all $v\in\texttt{memOps}(C)$ such that $R(v,\ell)\vee W(v,\ell)$, the **transactional last writer** of v according to $\mathcal{S}$, denoted $\mathcal{X}_{\mathcal{S}}(v)$, is the unique $u\in\texttt{memOps}(C)\cup\{\texttt{begin}(C)\}$ that satisfies four conditions:*

1. *$W(u,\ell)$,*
2. *$u<_{\mathcal{S}}v$,*
3. *$\neg(uHv)$, and*
4. *$\forall w\,(W(w,\ell)\wedge(u<_{\mathcal{S}}w<_{\mathcal{S}}v))\Rightarrow wHv$.*

The first two conditions for the transactional last-writer function $\mathcal{X}$ are the same as for the last-writer function $\mathcal{L}$. The third and fourth conditions of Definition 5 parallel the third condition of Definition 3, except that now v ignores vertices u or w that write to ℓ but which are hidden from v.

Sequential consistency can now be defined for computations that include transactions. The definition is exactly like Definition 4, except that the last-writer function $\mathcal{L}_{\mathcal{S}}$ is replaced by the transactional last-writer function $\mathcal{X}_{\mathcal{S}}$.

DEFINITION 6. ***Transactional sequential consistency** is the memory model*

$$TSC=\{(C,\Phi)\in\mathcal{U}:\exists\mathcal{S}\in\texttt{topo}(G(C))\text{ s.t. }\Phi=\mathcal{X}_{\mathcal{S}}\}\ .$$

4. TRANSACTIONAL MEMORY MODELS

In this section, we use our framework to define three different transactional memory models: serializability, race freedom, and prefix-race freedom. The intuition behind all three memory models is to find a single linear order $\mathcal{S}$ on all operations that both "explains" all memory operations and provides guarantees about every transaction. Serializability requires that all transactions appear as contiguous in $\mathcal{S}$. Race freedom weakens serializability by allowing transactions that do not "conflict" to interleave their memory operations in $\mathcal{S}$. Finally, prefix-race freedom weakens race freedom by only prohibiting conflicts with the prefix of a transaction.

Serializability

Serializability [22] is the standard correctness condition for transactional systems.

DEFINITION 7. *The **serializability** transactional memory model, ST, is the set of all traces $(C,\Phi)\in\mathcal{U}$ for which there exists a topological sort $\mathcal{S}\in\texttt{topo}(G(C))$ that satisfies two conditions:*

1. *$\Phi=\mathcal{X}_{\mathcal{S}}$, and*
2. *$\forall T\in\texttt{xactions}(C)$ and $\forall v\in V(C)$, we have $\texttt{begin}(T)\le_{\mathcal{S}}v\le_{\mathcal{S}}\texttt{end}(T)$ implies $v\in V(T)$).*

Informally, an execution belongs to ST if there exists an ordering on all operations $\mathcal{S}$ such that the observer function Φ is the transactional last writer $\mathcal{X}_{\mathcal{S}}$, and for every transaction T, the vertices in $V(T)$ appear contiguous in $\mathcal{S}$.

Race freedom

Our definition of race freedom is motivated by the observation that actual TM implementations allow independent transactions to interleave their executions provided that one transaction does not try to write to a memory location accessed by the other transaction. Normally, with only closed-nested transactions and ignoring operations from aborted transactions, we expect to be able to rearrange any interleaved execution order allowed by race freedom into an equivalent serializable order. As we shall see in Section 5, the two models are indeed equivalent for computations having only closed and committed transactions. With aborted and open transactions in the model, however, we shall discover that the models are distinct.

To define race freedom, we first describe what it means to have a transactional race between a memory operation and a transaction with respect to a topological sort of the computation dag.

DEFINITION 8. *Let C be a computation tree, and suppose that $\mathcal{S}\in\texttt{topo}(G(C))$ is a topological sort of $G(C)$. A **(transactional) race** with respect to $\mathcal{S}$ occurs between $v\in V(C)$ and $T\in\texttt{xactions}(C)$, denoted by the predicate $\textsc{Race}_{\mathcal{S}}(v,T)$, if $v\notin V(T)$ and there exists a $w\in\texttt{content}(T)$ satisfying the following conditions:*

1. *$\neg\,(vHw)$,*
2. *$\exists\ell\in\mathcal{M}$ s.t. $(R(v,\ell)\wedge W(w,\ell))\vee(W(v,\ell)\wedge R(w,\ell))\vee(W(v,\ell)\wedge W(w,\ell))$, and*
3. *$\texttt{begin}(T)\le_{\mathcal{S}}v\le_{\mathcal{S}}\texttt{end}(T)$.*

The notion of a race is easier to understand when all transactions are committed, in which case no vertices are hidden from each other. Intuitively, a race occurs between transaction T and a vertex $v\notin V(T)$ appearingbetween $\texttt{begin}(T)$ and $\texttt{end}(T)$ in $\mathcal{S}$ if v "conflicts" with some vertex $u\in\texttt{content}(T)$, where by "conflicts," we mean that v writes to a location that u reads or writes, or vice versa.

We can now define race freedom.

DEFINITION 9. *The **race-free** transactional memory model RFT is the set of all traces $(C,\Phi)\in\mathcal{U}$ for which there exists a topological sort $\mathcal{S}\in\texttt{topo}(G(C))$ satisfying two conditions:*

1. *$\Phi=\mathcal{X}_{\mathcal{S}}$, and*
2. *$\forall v\in V(C)$ and $\forall T\in\texttt{xactions}(C)$, $\neg\textsc{Race}_{\mathcal{S}}(v,T)$.*

The first condition of race freedom is the same as for serializability, that the observer function is the transactional last writer. The second condition allows an operation v to appear between $\texttt{begin}(T)$ and $\texttt{end}(T)$ in $\mathcal{S}$, but only provided no race between v and T exists.

Prefix-race freedom

The notion of a prefix-race is motivated by the operational semantics of TM systems. As two transactions T and T' execute, if T' discovers a memory-access conflict between a vertex $v\in T'$ and T, then the conflict must be with a vertex in T that has already executed, that is, the prefix of T that executes before v. For prefix-race freedom, no such conflicts may occur.

DEFINITION 10. *Let C be a computation tree, and let $\mathcal{S}\in\texttt{topo}(G(C))$ be a topological sort of $G(C)$. A **(transactional) prefix-race** with respect to $\mathcal{S}$ occurs between $v\in V(C)$ and $T\in\texttt{xactions}(C)$, denoted by the predicate $\textsc{PRace}_{\mathcal{S}}(v,T)$, if $v\notin V(T)$ and there exists a $w\in\texttt{content}(T)$ satisfying the following conditions:*

1. *$\neg(vHw)$*

2. $\exists \ell \in \mathcal{M}$ s.t. $(R(v,\ell) \wedge W(w,\ell)) \vee (W(v,\ell) \wedge R(w,\ell)) \vee (W(v,\ell) \wedge W(w,\ell))$.
3. $\texttt{begin}(T) \leq_{\mathcal{S}} w <_{\mathcal{S}} v \leq_{\mathcal{S}} \texttt{end}(T)$.

Thus, this definition is identical to Definition 8, except that the potential conflicting vertex w must occur before v in $\mathcal{S}$.

The notion of a prefix-race gives rise to an corresponding memory model in which prefix-races are absent.

DEFINITION 11. *The **prefix-race-free** transactional memory model $PRFT$ is the set of all traces $(C, \Phi) \in \mathcal{U}$ for which there exists a topological sort $\mathcal{S} \in \texttt{topo}(G(C))$ satisfying two conditions:*

1. *$\Phi = \mathcal{X}_{\mathcal{S}}$, and*
2. *$\forall v \in V(C)$ and $\forall T \in \texttt{xactions}(C)$, $\neg\text{PRACE}_{\mathcal{S}}(v, T)$.*

Thus, prefix-race freedom describes a weaker model than race freedom, where a vertex v is only guaranteed to not to conflict with the vertices of transaction T that appear before v in $\mathcal{S}$. If a "nontransactional" leaf node $v \in \texttt{memOps}(C)$ runs in parallel with a transaction T, all of Definitions 7, 9, and 11 check whether v interleaves within T's execution. Thus, these models can be thought of as guaranteeing "strong atomicity" in the parlance of Blundell, Lewis, and Martin [2]. In Scott's model [24], $\text{RACE}_{\mathcal{S}}(v, T)$ and $\text{PRACE}_{\mathcal{S}}(v, T)$ can be viewed as particular "conflict functions."

Relationships among the models

The following theorem shows that the memory models as presented are progressively weaker.

THEOREM 1. *$ST \subseteq RFT \subseteq PRFT$.*

PROOF. Follows directly from Definitions 7, 9, and 11. □

For computations with only closed and committed transactions, prefix-race freedom and serializability are equivalent, as we shall see in Section 5. When open and aborted transactions are considered, all three models are distinct.

5. DISTINCTNESS OF THE MODELS

In this section, we study the memory models of serializability, prefix-race freedom, and race freedom. Specifically, we show that for computations containing only committed and closed transactions, all three models are equivalent. We also demonstrate that when aborted and/or open transactions are allowed, all three models are distinct.

Dependency graphs

Before addressing the distinctness of the memory models directly, we first present an alternative characterization of sequential consistency for the special case of computations with only committed transactions. The idea of a "dependency" graph is to add edges to the computation dag to reflect the dependencies imposed by the observer function.

DEFINITION 12. *The set of **dependency** edges of a trace $(C, \Phi) \in \mathcal{U}_{\text{com}}$ is $\Psi_d(C, \Phi) = \{(u, v) \in V(C) \times V(C) : u = \Phi(v)\}$, and the set of **antidependency** edges is $\Psi_a(C, \Phi) = \{(u, v) \in V(C) \times V(C) : (\Phi(u) = \Phi(v)) \wedge W(v, \ell)\}$. The **dependency graph** of (C, Φ) is the graph $\mathcal{DG}(C, \Phi) = (V, E)$, where $V = V(C)$ and $E = E(C) \cup \Psi_d(C, \Phi) \cup \Psi_a(C, \Phi)$.*

The sets Ψ_d and Ψ_a capture the usual notions of dependency and antidependency edges from the study of compilers [13]. A dependency edge (u, v) indicates that v observed the value written by u. An antidependency edge (u, v) means that if both u and v observe the same write to a location ℓ, and if v performs a write, then u must "come before" v.

The following lemma, presented without proof, shows that in the universe of all traces with only committed transactions, a trace (C, Φ) is sequentially consistent if and only if the dependency graph $\mathcal{DG}(C, \Phi)$ is acyclic.[5]

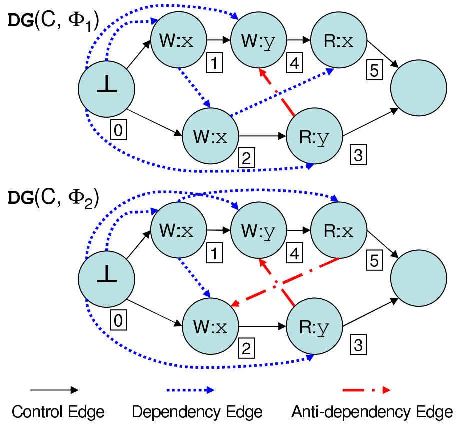


FIGURE 7: Dependency graphs $\mathcal{DG}(C, \Phi_1)$ and $\mathcal{DG}(C, \Phi_2)$ for the traces from Figure 6. Since $(C, \Phi_1) \in SC$, the graph $\mathcal{DG}(C, \Phi_1)$ is acyclic, but since $(G, \Phi_2) \notin SC$, the graph $DG(C, \Phi_2)$ contains a cycle, namely $\langle 2, 3, 4, 5, 2 \rangle$.

LEMMA 2. *Suppose that $(C, \Phi) \in \mathcal{U}_{\text{com}}$. Then, we have $(C, \Phi) \in SC$ if and only if the dependency graph $\mathcal{DG}(C, \Phi)$ is acyclic.* □

Figure 7 shows the dependency graphs for the example traces from Figure 6. Whereas the trace (C, Φ_1) is sequentially consistent, the trace (C, Φ_2) is not. Equivalently by Lemma 2, the dependency graph $\mathcal{DG}(C, \Phi_1)$ is acyclic, but the graph $\mathcal{DG}(C, \Phi_2)$ is not.

We can now prove the equivalence of serializability, race freedom, and prefix-race freedom when we consider only computations with committed and closed transactions.

THEOREM 3. *$ST \cap \mathcal{U}_{\text{c\&c}} = RFT \cap \mathcal{U}_{\text{c\&c}} = PRFT \cap \mathcal{U}_{\text{c\&c}}$.*

PROOF. Since Theorem 1 shows that $ST \subseteq RFT \subseteq PRFT$, it suffices to prove that $PRFT \cap \mathcal{U}_{\text{c\&c}} \subseteq ST \cap \mathcal{U}_{\text{c\&c}}$.

We start by defining some terminology. For $u, v \in V(C)$, define the ***alternation count*** of u and v as

$$A(u, v) = |\texttt{h}(u)| + |\texttt{h}(v)| - 2\,|\texttt{h}(\texttt{LCA}(u, v))| \ .$$

(The holders function $\texttt{h}$ was defined in Section 3.) Thus, $A(u, v)$ counts the number of transactions $T \in \texttt{xactions}(C)$ that contain either u or v, but not both. For any topological sort $\mathcal{S}$ of $G(C)$, define the ***alternation count*** of $\mathcal{S}$, denoted $alt(\mathcal{S})$, as the sum of all $A(u, v)$ for consecutive u and v in $\mathcal{S}$. Intuitively, $alt(\mathcal{S})$ counts the number of times we "switch" between transactions as we run through $\mathcal{S}$.

We prove by contradiction that for any trace $(C, \Phi) \in \mathcal{U}_{\text{c\&c}}$, we have $(C, \Phi) \in PRFT$ implies $(C, \Phi) \in SC$. Suppose that a trace $(C, \Phi) \in \mathcal{U}_{\text{c\&c}}$ exists that is prefix-race free but not serializable. Consider any prefix-race-free topological sort $\mathcal{S} \in \texttt{topo}(\mathcal{DG}(C, \Phi))$ that has a minimum alternation count $alt(\mathcal{S})$ over all sorts in $\texttt{topo}(\mathcal{DG}(C, \Phi))$. By Lemma 2, $\mathcal{S}$ satisfies the condition $\Phi = \mathcal{X}_{\mathcal{S}}$ (the first condition for all three transactional models).

Since $(C, \Phi) \notin ST$, some transaction T exists that is not contiguous in $\mathcal{S}$ (and therefore violates the second condition in Definition 7). Let T be such a transaction, and let v_1 be the first vertex such that $v_1 \notin V(T)$ and $\texttt{begin}(T) <_{\mathcal{S}} v <_{\mathcal{S}} \texttt{end}(T)$. Choose vertices $t <_{\mathcal{S}} u_1 \leq_{\mathcal{S}} u_2 <_{\mathcal{S}} v_1 \leq_{\mathcal{S}} v_2 <_{\mathcal{S}} w_1 <_{\mathcal{S}} w_2$,

[5] One must extend the definition of an antidependency edge to prove an analogous result when the computation C has aborted transactions. Lemma 2 does not hold without the assumption that every write to a location also performs a read.

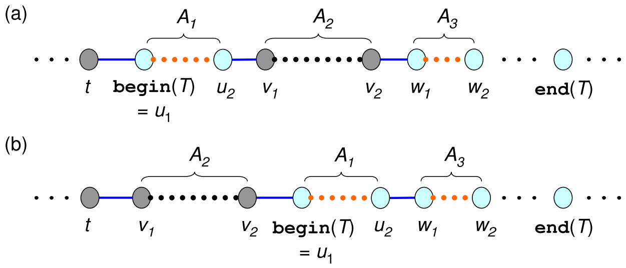


FIGURE 8: Two topological sorts of a computation graph $G(C)$ for a hypothetical trace (C, Φ) which is prefix-race free, but not serializable. Transaction T is not contiguous in the topological sort $\mathcal{S}$ in (a). One can convert $\mathcal{S}$ into the topological sort $\mathcal{S}'$ in (b). Doing so reduces the alternation count.

such that $u_1 = \texttt{begin}(T)$ as shown in Figure 8(a). Define the sets A_1, A_2, and A_3 as follows:

$$\begin{aligned} A_1 &= \{x \in V(T) : u_1 \leq_{\mathcal{S}} x \leq_{\mathcal{S}} u_2\}, \\ A_2 &= \{x \in V(C) - V(T) : v_1 \leq_{\mathcal{S}} x \leq_{\mathcal{S}} v_2\}, \text{ and} \\ A_3 &= \{x \in V(T) : w_1 \leq_{\mathcal{S}} x \leq_{\mathcal{S}} w_2\} \ . \end{aligned}$$

Define two sets $A_1 = \{x \in V(T) : u_1 \leq_{\mathcal{S}} x \leq_{\mathcal{S}} u_2\}$ and $A_3 = \{x \in V(T) : w_1 \leq_{\mathcal{S}} x \leq_{\mathcal{S}} w_2\}$ whose vertices all belong to $V(T)$. Define $A_2 = \{x \in V(C) - V(T) : v_1 \leq_{\mathcal{S}} x \leq_{\mathcal{S}} v_2\}$ as the set interleaved between the contiguous fragments of T.

From $\mathcal{S}$, we construct the new order $\mathcal{S}'$ shown in Figure 8(b) in which the intervals A_1 and A_2 are interchanged. We shall show that (1) $\mathcal{S}' \in \texttt{topo}(\mathcal{DG}(C, \Phi))$ (and therefore $\Phi = \mathcal{X}_{\mathcal{S}'}$), (2) $\mathcal{S}'$ is still a prefix-race-free topological sort of $\mathcal{DG}(C, \Phi)$, and (3) $alt(\mathcal{S}') < alt(\mathcal{S})$, thereby obtaining the contradiction that $\mathcal{S}$ is not a prefix-race-free topological sort with minimum alternation count.

To prove these three facts, we shall use a "nonconflicting" property: no pair of vertices $y \in A_1$ and $z \in A_2$ exist such that y and z access the same memory location and one of them is a write. Otherwise we have $\text{PRACE}_{\mathcal{S}}(z, T)$ by definition because $y \in \texttt{content}(T)$, $z \notin V(T)$, and $\texttt{begin}(T) <_{\mathcal{S}} y <_{\mathcal{S}} z <_{\mathcal{S}} \texttt{end}(T)$. Thus, A_1 and A_2 do not perform "conflicting" accesses to memory.

To establish (1), that $\mathcal{S}' \in \texttt{topo}(\mathcal{DG}(C, \Phi))$, we show that for any $y \in A_1$ and $z \in A_2$, no edge (y, z) belongs to the graph $\mathcal{DG}(C, \Phi)$. If we have $(y, z) \in \Psi_d(C, \Phi) \cup \Psi_a(C, \Phi)$, then y and z access the same memory location and one of those accesses is a write, contradicting the nonconflicting property above. Alternatively, if we have $(y, z) \in E(C)$, then $\texttt{LCA}(y, z)$ must be an S-node with y to the left of z. Since $z \notin V(T)$, we have $\texttt{LCA}(T, z)$ $(= \texttt{LCA}(y, z))$ is an S-node, and thus we have $\texttt{end}(T) \prec z$. Thus, $\mathcal{S}$ was not a valid sort of $\mathcal{DG}(C, \Phi)$, and $(y, z) \notin E(C)$.

To establish (2), that $\mathcal{S}'$ is prefix-race free, we show that swapping A_1 and A_2 cannot introduce any prefix races that weren't already there in $\mathcal{S}$. Suppose that there is a prefix-race in $\mathcal{S}'$. Then, there must exist a $v \in V(C)$ and a transaction $T_1 \in \texttt{xactions}(C)$ satisfying all three conditions of Definition 10 for $\mathcal{S}'$. Let $w \in \texttt{content}(T_1)$ be the candidate vertex that satisfies the three conditions. In particular, the third condition gives us $\texttt{begin}(T_1) <_{\mathcal{S}'} w <_{\mathcal{S}'} v <_{\mathcal{S}'} \texttt{end}(T_1)$. We consider two cases, each of which leads to a contradiction.

In the first case, suppose that $v <_{\mathcal{S}} w$. Since v and w swap in the two orders, we must have $v \in A_1$ and $w \in A_2$. But, then they conflict by the second condition of Definition 10, which cannot occur because of the nonconflicting property above.

In the second case, suppose that $w <_{\mathcal{S}} v$. Since there is no prefix-race in $\mathcal{S}$, the only situation in which this can happen is when v falls entirely outside transaction T_1 in $\mathcal{S}$, which is to say that $\texttt{begin}(T_1) <_{\mathcal{S}} w <_{\mathcal{S}} \texttt{end}(T_1) <_{\mathcal{S}} v$. Since $\texttt{end}(T_1)$ and v swapped, we must have $\texttt{end}(T_1) \in A_1$ and $v \in A_2$. Since $A_1 \subseteq \texttt{content}(T)$, it follows that $\texttt{end}(T_1) \in \texttt{content}(T)$, and thus T_1 must be nested within T. Consequently, we have $w \in A_1$, which cannot occur because of the nonconflicting property.

To establish (3), that $alt(\mathcal{S}') < alt(\mathcal{S})$, let us examine the difference $\delta = alt(\mathcal{S}) - alt(\mathcal{S}')$ in the alternation counts of $\mathcal{S}$ and $\mathcal{S}'$. The only terms that contribute to δ are at the boundaries of A_1 and A_2. We have that

$$\begin{aligned} \delta &= A(t, u_1) + A(u_2, v_1) + A(v_2, w_1) \\ &\quad - A(t, v_1) - A(v_2, u_1) - A(u_2, w_1) \\ &= 2\,(|\texttt{h}(\texttt{LCA}(t, v_1))| + |\texttt{h}(\texttt{LCA}(v_2, u_1))| \\ &\quad + |\texttt{h}(\texttt{LCA}(u_2, w_1))| - |\texttt{h}(\texttt{LCA}(t, u_1))| \\ &\quad - |\texttt{h}(\texttt{LCA}(u_2, v_1))| - |\texttt{h}(\texttt{LCA}(v_2, w_1))|) \ . \end{aligned}$$

By construction, we know that $\{u_1, u_2, w_1, w_2\} \subseteq V(T)$, whereas none of t, v_1, and v_2 have T as an ancestor. For any $y \in V(T)$ and $z \notin V(T)$, we have $\texttt{LCA}(y, z) = \texttt{LCA}(T, z)$, which yields

$$\begin{aligned} \delta &= 2\,(|\texttt{h}(\texttt{LCA}(t, v_1))| + |\texttt{h}(\texttt{LCA}(u_2, w_1))| \\ &\quad - |\texttt{h}(\texttt{LCA}(t, T))| - |\texttt{h}(\texttt{LCA}(T, v_1))|) \ . \end{aligned}$$

Since $\texttt{LCA}(u_2, w_1) \in \texttt{desc}(T)$, we know $\texttt{h}(\texttt{LCA}(u_2, w_1)) \supseteq \texttt{h}(T)$ and $|\texttt{h}(\texttt{LCA}(u_2, w_1))| \geq |\texttt{h}(T)|$. Since $t, v_1 \notin V(T)$, we have $\texttt{h}(\texttt{LCA}(T, t)) \subset \texttt{h}(T)$ and $\texttt{h}(\texttt{LCA}(T, v_1)) \subset \texttt{h}(T)$.[6] Thus,

$$|\texttt{h}(\texttt{LCA}(u_2, w_1))| > \max\{|\texttt{h}(\texttt{LCA}(T, t))|, |\texttt{h}(\texttt{LCA}(T, v_1))|\} \ ,$$

and a similar algebra yields

$$|\texttt{h}(\texttt{LCA}(t, v_1))| \geq \min\{|\texttt{h}(\texttt{LCA}(T, t))|, |\texttt{h}(\texttt{LCA}(T, v_1))|\} \ .$$

Consequently, we conclude that $\delta = alt(\mathcal{S}) - alt(\mathcal{S}') > 0$. □

Aborted transactions

We now consider computations with aborted transactions. We are unaware of any prior work on transactional semantics that explicitly models aborted transactions. The reason is simple: when computations have only closed transactions, aborted transactions do not affect a program's output. Since TM systems do not allow committed transactions to observe data directly from aborted transactions, in most cases, vertices from aborted transactions are free to observe arbitrary values.[7]

In a system with open nesting, however, we must include aborted transactions in the memory model if we wish to understand what happens when an open transaction commits but its parent aborts. We contend that a reasonable transactional consistency model for open transactions must not only model aborted transactions, but it should also guarantee that an aborted transaction T is consistent up to the point it aborts. Otherwise, any open subtransactions within T may obtain inconsistent values and still commit.

The next theorem shows that when aborted transactions are modeled, the three transactional memory models are distinct.

THEOREM 4. $ST \cap \mathcal{U}_{\text{clo}} \subsetneq RFT \cap \mathcal{U}_{\text{clo}} \subsetneq PRFT \cap \mathcal{U}_{\text{clo}}$.

PROOF. Since Theorem 1 shows that $ST \subseteq RFT \subseteq PRFT$, we need only show that $ST \cap \mathcal{U}_{\text{clo}} \neq RFT \cap \mathcal{U}_{\text{clo}}$ and that $RFT \cap \mathcal{U}_{\text{clo}} \neq PRFT \cap \mathcal{U}_{\text{clo}}$.

We first exhibit a computation that is race free but not serializable. Consider the computation dag G shown in Figure 9. Let (C_1, Φ_1) be the trace that generates G, where transactions

[6] In this case, we have a proper subset because $\texttt{LCA}(T, t), \texttt{LCA}(T, v_1) \in \texttt{pAnces}(T)$ and we exclude T.

[7] This intuition is not strictly true in a model that does not analyze an execution *a posteriori*, since control flow can be affected by inconsistent data and prevent a program from terminating.

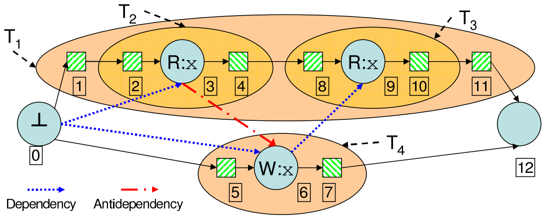


FIGURE 9: An example distinguishing the memory models. The transactions T_2 and T_3 are closed-nested inside of T_1. If transaction T_4 commits, then this computation is not serializable, because T_4 must interleave inside of T_1. If both transactions T_2 and T_3 abort, then the execution is race free. If T_2 aborts and T_3 commits, then this execution is not race free, but it is prefix-race free.

T_2 and T_3 abort but transaction T_4 commits. We shall show that $(C_1, \Phi_1) \in RFT$, but $(C_1, \Phi_1) \notin ST$.

If transaction T_4 commits, then for any topological sort $\mathcal{S}$ satisfying $\mathcal{X}_{\mathcal{S}} = \Phi$, we must have $0 <_{\mathcal{S}} 3 <_{\mathcal{S}} 6 <_{\mathcal{S}} 9$. Thus, T_1 cannot be contiguous within $\mathcal{S}$, implying that $(C_1, \Phi_1) \notin ST$.

We can show that (C_1, Φ_1) is race free, however. Let $\mathcal{S}$ be $\langle 0, 1, \ldots, 12 \rangle$. One can verify that Φ_1 is indeed the transactional last-writer function according to $\mathcal{S}$ (since T_4 commits, $\neg(6 H 9)$, and thus $\Phi_1(9) = \mathcal{X}_{\mathcal{S}}(9)$). The only transactions that might violate the second condition of Definition 9 are transactions that do not appear contiguous in $\mathcal{S}$, in this case, only T_1. The only candidate vertex v for $\text{RACE}_{\mathcal{S}}(v, T_1)$ is $v = 6$. Since T_2 is an aborted subtransaction of T_1, however, neither 3 or 9 belong to $\texttt{content}(T_1)$. Thus, picking $\mathcal{S} = \langle 0, 1, \ldots, 12 \rangle$ ensures that T_1 causes no races.

We next exhibit a computation that is prefix-race free but not race free. Consider (C_2, Φ_2) as the trace generating the same computation dag G from Figure 9, but this time with T_2 aborted and T_3 and T_4 committed. We shall show that $(C_2, \Phi_2) \notin RFT$, but that $(C_2, \Phi_2) \in PRFT$.

To show that (C_2, Φ_2) is not race free, observe that in any topological sort $\mathcal{S} \in \texttt{topo}(G)$ for which $\Phi = \mathcal{X}_{\mathcal{S}}$, we must have $\text{RACE}_{\mathcal{S}}(6, T_1)$, since $\texttt{begin}(T_1) <_{\mathcal{S}} 6 <_{\mathcal{S}} \texttt{end}(T_1)$, vertices 6 and 9 access the same memory location x, and vertex 6 is a write, and $\neg(6 H 9)$. The order $\mathcal{S} = \langle 0, 1, \ldots, 12 \rangle$ is prefix-race free, however, since $9 \not<_{\mathcal{S}} 6$. The only transactions that might violate the second condition of prefix-race freedom are those that do not appear contiguous in $\mathcal{S}$, in this case, only T_1. When we look at the vertex $v = 6$ that falls between $\texttt{begin}(T_1)$ and $\texttt{end}(T_1)$, we only look at the prefix of T_1 before v (vertices 1 through 4) for a prefix-race conflict, and there is none.

The proof holds whether T_1 commits or aborts. □

Open transactions

We now study computations with open transactions but where all transactions commit. In this context, the three models ST, RFT, and $PRFT$ are distinct.

THEOREM 5. $ST \cap \mathcal{U}_{\text{com}} \subsetneq RFT \cap \mathcal{U}_{\text{com}} \subsetneq PRFT \cap \mathcal{U}_{\text{com}}$.

PROOF. Since Theorem 1 shows that $ST \subseteq RFT \subseteq PRFT$, we need only show that $ST \cap \mathcal{U}_{\text{com}} \neq RFT \cap \mathcal{U}_{\text{com}}$ and that $RFT \cap \mathcal{U}_{\text{com}} \neq PRFT \cap \mathcal{U}_{\text{com}}$. The trace in Figure 10 shows a $(C_1, \Phi_1) \notin ST$, but $(C_1, \Phi_1) \in RFT$. Figure 11 shows $(C_2, \Phi_2) \notin RFT$, but $(C_2, \Phi_2) \in PRFT$. □

Trade-offs among the models

The three transactional memory models of serializability, race freedom, and prefix-race freedom exhibit different behaviors in TM systems that have open transactions.

With serializability, for any trace $(C, \Phi) \in ST$, we can "change" the trace to convert any open transaction T' nested inside a committed transaction T from open to closed while still keeping the same Φ, and still be serializable. Thus, in some sense, with serializability, open nesting only differs from closed nesting if an open transaction commits, but its parent aborts.

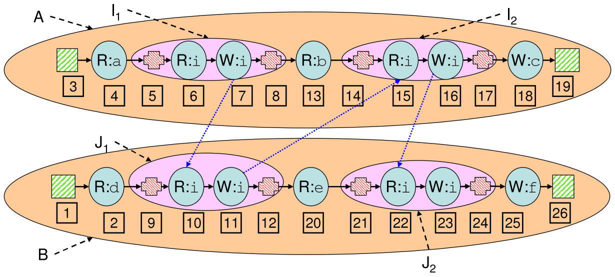


FIGURE 10: When all transactions commit, this computation dag $G(C_1)$ with observer edges Φ_1 is not serializable, but is race free. This trace represents Schedule 3 from the program in Figure 2.

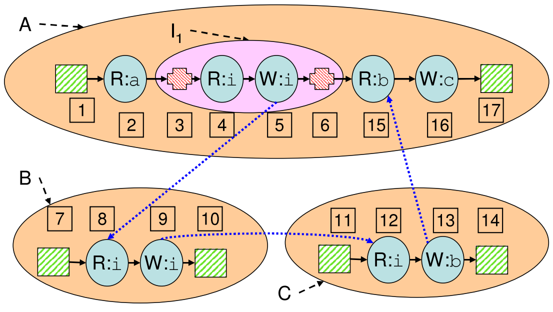


FIGURE 11: When all transactions commit, this computation dag $G(C_2)$ with observer edges Φ_2 is prefix-race free, but not race free, because a race exists between vertices 13 and 15.

Race-freedom appears to be more difficult to implement than either serializability or prefix race-freedom. For example, consider the example from Figures 3 and 11. After an transaction I_1 (open-nested in A) commits, any number of other transactions (B and C) can read values written by that open transaction and commit their changes, all before the original outer transaction A completes. To support race freedom, it seems we may need to maintain the footprints of B and C even after they have committed to detect a future conflict with A.

6. THE *ON* OPERATIONAL MODEL

This section presents an abstract operational model for open nesting, called the *ON* model, which is a generalization of the Stanford model [16]. We prove that the *ON* model implements at least prefix race-freedom but is strictly weaker than race freedom.

We begin our description of the *ON* model by defining some notation. For any set $S \subseteq \texttt{nodes}(C)$ of tree nodes, let $\texttt{lowest}(S)$ be the node $X \in S$ such that $S \subseteq \texttt{ances}(X)$, if such a X exists. Otherwise, define $\texttt{lowest}(S) = \texttt{null}$. Thus, if all nodes in S all fall on one root-to-leaf path in C, then $\texttt{lowest}(S)$ is the lowest node on that path. Define $\texttt{highest}(S)$ in a similar fashion. For any $T \in \texttt{xactions}(C)$, define $\texttt{xparent}(T) = \texttt{lowest}(\texttt{ances}(T) \cap \texttt{xactions}(C))$, that is, $\texttt{xparent}(T)$ is the transactional parent of T. For any $X \in \texttt{nodes}(C)$, let $\texttt{xAnces}(X) = \texttt{ances}(X) \cap \texttt{xactions}(C)$ be the set of transactional ancestors of X.

Abstractly, we shall view the *ON* model for open nesting as a nondeterministic state machine *ON* that constructs a sequence of traces. The initial trace contains a computation tree consisting of a single S-node $\texttt{root}(C) \in \texttt{spNodes}(C)$ with associated sets $\texttt{xactions}(C) = \{\texttt{root}(C)\}$ and $\texttt{open}(C) = \texttt{committed}(C) = \texttt{aborted}(C) = \emptyset$ and an empty observer function Φ. By assuming that $\texttt{root}(C) \in \texttt{xactions}(C)$, we simplify the description of the model by treating the entire computation C as a global closed transaction in which other transactions are nested. The computation also maintains an initially empty auxiliary set $\texttt{done}(C) \subseteq \texttt{nodes}(C)$ of nodes that have finished their execution. The computation tree C and all these associated sets only grow during the execution.

At any time during the computation, a subset $\texttt{ready}(C)$ of S-nodes are designated as ***ready***, meaning that they can issue a program instruction, which include `read`, `write`, `fork`, `join`, `xbegin`, `xbegin_open`, and `xend`. The ON machine nondeterministically chooses a ready S-node to issue an instruction, and the machine processes the instruction which augments (C, Φ) by adding nodes to the tree and to its associated sets. Unlike other associated sets $\texttt{ready}(C)$ may grow and shrink during execution.

We shall factor the description of the state machine ON by describing the creation of the computation tree C and the observer function Φ separately.

Creating the computation tree

How the computation tree C evolves depends on the instructions that are issued nondeterministically. Let X be the S-node that issues an instruction. The instructions are handled as follows:

- `read` from a location $\ell \in \mathcal{M}$: If the read causes a conflict (more about conflicts when we describe the creation of the observer function) with one or more transactions, abort[8] the deepest such transaction T by adding all transactions $T' \in \texttt{desc}(T) \cap \texttt{xactions}(T) - \texttt{done}(C)$ both to $\texttt{aborted}(C)$ and to $\texttt{done}(C)$. Keep checking for and aborting conflicting transactions T, deepest to shallowest, until no such conflicting transactions exist. Then, create a new `read` node $v \in \texttt{memOps}(C)$ as the last child of the S-node X. Add v to $\texttt{done}(C)$.
- `write` to a location $\ell \in \mathcal{M}$: Similar to `read`.
- `fork`: Create a new P-node $Y \in \texttt{nodes}(C)$ as a child of X, and create two new S-nodes as children of Y. Add these two children to $\texttt{ready}(C)$, and remove X from $\texttt{ready}(C)$.
- `join`: Test whether X's sibling belongs to $\texttt{done}(C)$. If yes, then add X and then $\texttt{parent}(X)$ to $\texttt{done}(C)$. Remove X from $\texttt{ready}(C)$, and add $\texttt{parent}(\texttt{parent}(X))$ (the grandparent of X which is an S-node) to $\texttt{ready}(C)$. If no, then remove X from $\texttt{ready}(C)$, and add X to $\texttt{done}(C)$.
- `xbegin`: Create a new S-node $Y \in \texttt{nodes}(C)$ as the last child of X. Add Y to $\texttt{xactions}(C)$. Remove X from $\texttt{ready}(C)$, and add Y to $\texttt{ready}(C)$.
- `xbegin_open`: Similar to `xbegin`, but also add Y to $\texttt{open}(C)$.
- `xend`: Test whether $X \in \texttt{xactions}(C)$. If yes, remove X from $\texttt{ready}(C)$, and add $\texttt{parent}(X)$ to $\texttt{ready}(C)$. Add X to $\texttt{done}(C)$ and to $\texttt{committed}(C)$. If no, error.

The ON machine maintains several invariants. All transactions are S-nodes. Every P-node has an S-node as its parent and has exactly two S-nodes as children. If an S-node is ready, none of its ancestors are ready.

Creating the observer function

To create the observer function, the ON model maintains auxiliary state to keep track of how values are propagated among transactions and global memory. Specifically, every transaction $T \in \texttt{xactions}(C)$ maintains a ***readset*** $\texttt{R}(T)$ and a ***writeset*** $\texttt{W}(T)$. The readset $\texttt{R}(T)$ is a set of pairs (ℓ, v), where $\ell \in \mathcal{M}$ is a memory location and $v \in \texttt{memOps}(C)$ is the memory operation that read from ℓ, that is, we maintain the invariant $R(v, \ell)$ for all $(\ell, v) \in \bigcup_{T \in \texttt{xactions}(C)} \texttt{R}(T)$. The writeset $\texttt{W}(T)$ is similarly defined. We initialize $\texttt{R}(\texttt{root}(C)) = \texttt{W}(\texttt{root}(C)) = \{(\ell, \texttt{begin}(\texttt{root}(C))) : \ell \in \mathcal{M}\}$.

The ON model maintains two invariants concerning readsets and writesets. First, it maintains $\texttt{W}(T) \subseteq \texttt{R}(T)$ for every transaction $T \in \texttt{xactions}(C)$, that is, a write to a location also counts as a read to that location. Second, $\texttt{R}(T)$ and $\texttt{W}(T)$ each contain at most one pair (ℓ, v) for any location ℓ. Because of this second invariant, we employ the shorthand $\ell \in \texttt{R}(T)$ to mean that there exists a node u such that $(\ell, u) \in \texttt{R}(T)$, and similarly for $\texttt{W}(T)$. We also overload the union operator to accommodate this assumption: if we write $\texttt{R}(T) \leftarrow \texttt{R}(T) \cup \{(\ell, u)\}$, then if there exists $(\ell, u') \in \texttt{R}(T)$, we mean to replace it with (ℓ, u). Likewise, if u accesses a location ℓ, we employ the shorthand $u \in \texttt{R}(T)$ to mean that $(\ell, u) \in \texttt{R}(T)$, and similarly for $\texttt{W}(T)$.

The state machine ON handles events as follows, where X is the S-node that issues the instruction:

- `read` from location $\ell \in \mathcal{M}$: If there exists a $T \in \texttt{xactions}(C) - \texttt{done}(C) - \texttt{ances}(X)$ such that $\ell \in \texttt{W}(T)$, then a conflict occurs. Let v be the read operation added as the last child of X. Define $S_\ell = \{T \in \texttt{xactions}(C) \cap \texttt{ances}(v) : \ell \in \texttt{R}(T)\}$, let $T' = \texttt{lowest}(S_\ell)$, and let $(\ell, u) \in \texttt{R}(T')$. Add (ℓ, u) to $\texttt{R}(T)$, and set $\Phi(v) = u$.
- `write` to a location $\ell \in \mathcal{M}$: Similar to `read`, but to check for a conflict, test whether there exists a $T \in \texttt{xactions}(C) - \texttt{done}(C) - \texttt{ances}(X)$ such that $\ell \in \texttt{R}(T)$. Find u in the same way, and add (ℓ, u) both to $\texttt{R}(T)$ and to $\texttt{W}(T)$, and set $\Phi(v) = u$.
- `xbegin` and `xbegin_open`: Initialize $\texttt{R}(Y) = \emptyset$ and $\texttt{W}(Y) = \emptyset$.
- `xend`: If $X \in \texttt{closed}(C)$, then add $\texttt{R}(X)$ to $\texttt{R}(\texttt{xparent}(X))$ and add $\texttt{W}(X)$ to $\texttt{W}(\texttt{xparent}(X))$. If $X \in \texttt{open}(C)$, then let $Q = \texttt{xAnces}(T)$. For any $(\ell, u) \in \texttt{W}(T)$, let $\alpha_\ell = \{T' \in Q \mid \ell \in \texttt{R}(T')\}$. For all such $T' \in \alpha_\ell$, $\texttt{R}(T') \leftarrow \texttt{R}(T') \cup \{(\ell, u)\}$. Similarly, let $\beta_\ell = \{T' \in Q \mid \ell \in \texttt{W}(T')\}$. For all $T' \in \beta_\ell$, $\texttt{W}(T') \leftarrow \texttt{W}(T') \cup \{(\ell, u)\}$.
- `fork` or `join`: No action.

The Stanford model [16] is similar to the ON model, except that it only supports "linear" nesting (transactions can have no parallel transactions within them) and the choice of which transaction to abort is nondeterministic. Neither of these differences affects the theorems that deal with the ON model, assuming they implement their system with pessimistic concurrency control.

Prefix race-freedom of ON

We now prove that the ON model is prefix-race free with respect to the natural topological sort $\mathcal{S}$ of $G(C)$ created by the nondeterministic operation of the ON machine. Specifically, as the ON model generates a trace (C, Φ), it creates tree nodes $\texttt{nodes}(C) = \texttt{spNodes}(C) \cup \texttt{memOps}(C)$ and eventually marks these nodes as "done" by placing them in $\texttt{done}(C)$. We can view this process as determining the topological sort $\mathcal{S}$ of $G(C)$ as follows. When a node $X \in \texttt{nodes}$ is created, the vertex $\texttt{begin}(X) \in V(C)$ is appended to $\mathcal{S}$. When a node is marked as done, the vertex $\texttt{end}(X) \in V(C)$ is appended to $\mathcal{S}$. If the node X is a memory operation, we have $\texttt{begin}(X) = \texttt{end}(X) = X$, and we view it as being appended only once. It is straightforward to verify that $\mathcal{S}$ is indeed a topological sort of $G(C)$, and indeed of $\mathcal{DG}(C, \Phi)$.

We begin with a definition of time in the ON model. If $v \in V(C)$ is the tth element of $\mathcal{S}$, we say that v occurs at ***time*** t, and we write $t = \mathcal{S}(v)$. Thus, for all $u, v \in V(C)$, we have

[8] The ON machine uses a "pessimistic" concurrency control mechanism in that it immediately aborts a conflicting transaction T upon conflict. Moreover, it always aborts T rather than its own transaction. One could abort the transaction performing the `read`, but the model is simpler by always aborting T and not providing a nondeterministic choice.

$u \le_{\mathcal{S}} v$ if and only if $\mathcal{S}(u) \le \mathcal{S}(v)$. We can view the evolution of (C, Φ) over time as a sequence $(C^{(t)}, \Phi^{(t)})$ for $t = 0, 1, \ldots$, where the operation that occurs at time t creates $(C^{(t)}, \Phi^{(t)})$ from $(C^{(t-1)}, \Phi^{(t-1)})$. For convenience, however, we shall omit time indices unless clarity demands it.

We define two time-sensitive sets. The set of ***active*** transactions at any given time is $\mathtt{active}(C) = \mathtt{xactions}(C) - \mathtt{done}(C)$. The ***spine*** of a memory location $\ell \in \mathcal{M}$ at any given time is $\mathtt{spine}(\ell) = \{T \in \mathtt{active}(C) : \ell \in \mathtt{W}(T)\}$.

We now state a structural lemma that describes invariants of the computation tree C as it evolves.

Lemma 6. *The ON machine maintains the following invariants:*

1. *If $T \in \mathtt{active}(C)$, then we have $\mathtt{xAnces}(T) \subseteq \mathtt{active}(C)$.*
2. *If $v \in \mathtt{W}(T)$, then $v \in V(T)$.*
3. *All transactions in $\mathtt{spine}(\ell)$ are on the same root to leaf path in C, and hence the node $\mathtt{lowest}(\mathtt{spine}(\ell))$ exists.*
4. *If $\ell \in \mathtt{R}(T)$, where $T \in \mathtt{active}(C)$, then we have either $\mathtt{spine}(\ell) \subseteq \mathtt{ances}(T)$ or $T \in \mathtt{ances}(\mathtt{lowest}(\mathtt{spine}(\ell)))$.*
5. *If $(\ell, u) \in \mathtt{R}(T)$ for some $T \in \mathtt{active}(C)$, then $(\ell, u) \in \mathtt{W}(T')$, where $T' = \mathtt{lowest}(\mathtt{xAnces}(T) \cap \mathtt{spine}(\ell))$.*
6. *Let $(\ell, u) \in \mathtt{W}(T_1)$ and $(\ell, v) \in \mathtt{W}(T_2)$, where $T_1, T_2 \in \mathtt{spine}(\ell)$. If $T_1 \in \mathtt{ances}(T_2)$, then $u \le_{\mathcal{S}} v$.*
7. *Let $(\ell, u) \in \mathtt{W}(T)$ and let $u <_{\mathcal{S}} v$ such that $W(v, \ell)$. Then, we have $v \in \mathtt{desc}(T)$.*

Proof. Induction on time. □

The next three lemmas describe additional structure of the computation tree.

Lemma 7. *For all $T \in \mathtt{aborted}(C)$ and $T' \in \mathtt{active}(C)$, if $v \in \mathtt{content}(T)$, then we have $v \notin \mathtt{W}(T')$.* □

Lemma 8. *If $v \in \mathtt{memOps}(C)$ accesses $\ell \in \mathcal{M}$, then at time $\mathcal{S}(v)$, we have $\mathtt{spine}(\ell) \subseteq \mathtt{ances}(v)$.* □

Lemma 9. *For all $v \in V(C)$, $T \in \mathtt{aborted}(C)$, and $w \in \mathtt{content}(T)$, if $\mathtt{end}(T) <_{\mathcal{S}} v$, then we have wHv.* □

The next lemma shows that a memory location written within a transaction remains in the writeset of some active descendant of the transaction.

Lemma 10. *Let $w \in \mathtt{memOps}(C) \cap \mathtt{content}(T)$ be a memory operation in a transaction $T \in \mathtt{xactions}(C)$, and suppose that $W(w, \ell)$ for some location $\ell \in \mathcal{M}$. Then, at all times t in the range $\mathcal{S}(w) < t < \mathcal{S}(\mathtt{end}(T))$, we have $\ell \in \mathtt{W}(T')$ for some $T' \in \mathtt{desc}(T) \cap \mathtt{active}(T)$.*

Proof. We proceed by induction on time. For the base case, at time $\mathcal{S}(w)$, location ℓ is added to $\mathtt{W}(\mathtt{xparent}(w))$, and $\mathtt{xparent}(w) \in \mathtt{desc}(T) \cap \mathtt{active}(T)$. For the inductive step, let $\ell \in \mathtt{W}(T')$ for some $T' \in \mathtt{desc}(T) \cap \mathtt{active}(T)$. Once a location is added to a transaction's writeset, it is never removed until the transaction commits or aborts. If $T' = T$, then we are done. Otherwise, we have $T' \in \mathtt{pDesc}(T)$ and by definition of $\mathtt{content}(T)$, it follows that $T' \notin \mathtt{open}(C) \cup \mathtt{aborted}(C)$. Therefore, at time $\mathcal{S}(\mathtt{end}(T'))$, location ℓ is added to $\mathtt{W}(\mathtt{xparent}(T'))$, at which time $\mathtt{xparent}(T')$ is an active descendant of T. □

We can now prove that the ON model admits no prefix-races.

Lemma 11. *For all $v \in \mathtt{memOps}(C)$ and $T \in \mathtt{xactions}(C)$, we have $\neg\mathrm{PRACE}_{\mathcal{S}}(v, T)$.*

Proof. Suppose for contradiction that $\mathrm{PRACE}_{\mathcal{S}}(v, T)$. Then, by Definition 10, we have $v \notin V(T)$ (or equivalently, $T \notin \mathtt{ances}(v)$), and there exists a $w \in \mathtt{content}(T)$ such that $\neg(vHw)$ and $\mathtt{begin}(T) <_{\mathcal{S}} w <_{\mathcal{S}} v <_{\mathcal{S}} \mathtt{end}(T)$, where v and w access the same location $\ell \in \mathtt{memOps}(C)$ and one of those accesses is a write.

Consider the case when $W(u, \ell)$. By Lemma 10, at time $\mathcal{S}(v)$ we have $\ell \in \mathtt{W}(T')$, where $T' \in \mathtt{desc}(T)$. At time $\mathcal{S}(v)$, vertex v is added to $\mathtt{R}(\mathtt{xparent}(v))$, and $\mathtt{xparent}(v) \notin \mathtt{desc}(T')$, because otherwise $v \in \mathtt{desc}(T') \subseteq \mathtt{desc}(T)$. Therefore, at time $\mathcal{S}(v)$, we have $\ell \in \mathtt{R}(\mathtt{xparent}(v))$ and $\ell \in \mathtt{W}(T')$, which violates Invariant 4 in Lemma 6.

The case when $R(u, \ell)$ is analogous. □

The next series of lemmas show that the observer function created by the ON machine is the transactional last-writer function according to $\mathcal{S}$.

Lemma 12. *For all $T \in \mathtt{xactions}(C)$, $T' \in \mathtt{active}(C)$, and $u \in \mathtt{content}(T)$, if $T \notin \mathtt{committed}(C)$ at time t and $u \in \mathtt{W}(T')$ at time t, then $T' \in \mathtt{desc}(T)$.*

Proof Sketch. One can prove by induction that at any time t such that $\mathcal{S}(u) \le t < \mathcal{S}(\mathtt{end}(T))$, we have $\mathtt{h}(u) \subseteq \mathtt{xAnces}(u) - \mathtt{pAnces}(T)$ and $\mathtt{h}(u) \cap (\mathtt{open}(T) - \{T\}) = \emptyset$. □

Lemma 13. *For any $v \in \mathtt{memOps}(C)$, if $\Phi(v) = u$, then $\neg(uHv)$.*

Proof. Assume for contradiction that uHv holds. Then, there exists $T \in \mathtt{pDesc}(\mathtt{LCA}(u, v)) \cap \mathtt{aborted}(C)$ such that $u \in \mathtt{content}(T)$. If the ON machine sets $\Phi(v) = u$, then $u \in \mathtt{R}(T')$ for some $T' \in \mathtt{xAnces}(v)$. By Invariant 5 in Lemma 6, it follows that $u \in \mathtt{W}(T'')$, where $T'' \in \mathtt{ances}(T')$, and hence $T \in \mathtt{ances}(T'')$ by Lemma 12. Therefore, we have $T \in \mathtt{ances}(v)$, and $\mathtt{LCA}(u, v) = T \in \mathtt{pDesc}(T)$. Contradiction. □

We say that a vertex $v \in \mathtt{memOps}(C)$ is ***alive***, denoted $\mathtt{alive}(v)$, if $\mathtt{h}(v) \cap \mathtt{aborted}(C) = \emptyset$.

Lemma 14. *Let $w \in V(C)$ be the last vertex in $\mathcal{S}$ such that $W(w, \ell)$ and $\mathtt{alive}(w)$. Then, there exists $(T) \in \mathtt{spine}(\ell)$ such that $(\ell, w) \in \mathtt{W}(T')$.*

Proof Sketch. At time $\mathcal{S}(w)$, by Invariant 3 of Lemma 6, we have $(\ell, w) \in \mathtt{W}(\mathtt{xparent}(w))$ and $\mathtt{xparent}(w) \in \mathtt{spine}(\ell)$. Assume for contradiction that w is not on the spine. Since w is alive, w can only be removed from $\mathtt{spine}(\ell)$ by being overwritten by some y such that $W(y, \ell)$ holds, and $w <_{\mathcal{S}} y$ (from Invariant 6 from Lemma 6). Since w is the last writer to ℓ which is alive, we have $\neg\mathtt{alive}(y)$. One can show that $\neg\mathtt{alive}(w)$ in this case. □

Lemma 15. *For $u, v \in \mathtt{memOps}(C)$ that both access a memory location $\ell \in \mathcal{M}$, if $\Phi(v) = u$, then for any $w \in \mathtt{memOps}(C)$ such that $u \prec_{\mathcal{S}} w \prec_{\mathcal{S}} v$ and $W(w, \ell)$, we have wHv.*

Proof. Assume for the purpose of contradiction that there exists a $w \in \mathtt{memOps}(C)$ such that $u \prec_{\mathcal{S}} w \prec_{\mathcal{S}} v$, $W(w, \ell)$, and $\neg wHv$. Consider the last such w.

If $w \in \mathtt{content}(T)$ for some $T \in \mathtt{aborted}(C^{(\mathcal{S}(v))})$, then by Lemma 9 we have wHv.

If w is not in the contents of any aborted transaction at time $\mathcal{S}(v)$, then by Lemma 14, we have $w \in \mathtt{W}(T)$ for some transaction $T \in \mathtt{spine}(\ell)$ and $T \in \mathtt{ances}(v)$ by Lemma 8. Let $T_R = \mathtt{lowest}(\{T \in \mathtt{xAnces}(v) : \ell \in \mathtt{R}(T)\})$, and let $T_W = \mathtt{lowest}(\{T \in \mathtt{xAnces}(v) : \ell \in \mathtt{W}(T)\})$. If $\Phi(v) = u$, then we have $u \in \mathtt{R}(T_R)$, since the ON machine always reads from the lowest ancestor that has ℓ in its readset. By Invariant 5, we have $u \in \mathtt{W}(T_W)$, but since $u <_{\mathcal{S}} w$, we have $T_W \in \mathtt{pAnces}(T)$ by Invariant 6 in Lemma 6. Therefore, T is a lower ancestor of v than T_W, contradicting the fact that T_W is the lowest ancestor of v with ℓ in its writeset. □

We now can prove that the observer function for the ON model is the transactional last-writer function.

Lemma 16. *If the ON model generates an execution (C, Φ), then $\Phi = \mathcal{X}_{\mathcal{S}}$.*

Proof. Let $\Phi(v, \ell) = u$. To be the transactional last writer $\mathcal{X}_{\mathcal{S}}$, the observer function Φ must satisfy four conditions. The first two, $W(u, \ell)$ and $u \prec_{\mathcal{S}} v$, hold by the ON machine's operation. Lemmas 13 and 15 provide the last two conditions. □

THEOREM 17. *The ON model implements prefix race-free freedom.*

PROOF. Combine Lemmas 11 and 16. □

7. CONCLUSION

Open nesting provides a loophole in the strict serializability requirement for transactional programs, but at what cost to program understandability? When we began our study, we believed that open nesting could be modularized so that users of a subroutine would not need to know whether the subroutine uses open nesting. Unfortunately, as we saw in Section 2, Figure 4, implementing open nesting using prefix-race freedom can lead to unexpected program behavior if the programmer is unaware of the existence of open transactions in subroutines. Race-freedom admits similar anomalous behavior. At least at the level of memory semantics, it seems unlikely that such anomalous behaviors can be completely and safely hidden.

Our study leaves open the possibility, however, that open nesting can be modularized at the level of program semantics. Specifically, one may be able to devise a program semantics for open nesting, as discussed in [20], and formally relate it to a memory model in such a way that anomalies in the memory model do not propagate to the program model. For example, the anomalous memory semantics for open nesting provided by prefix-race freedom might be able to be hidden from programmers at a higher level without sacrificing the advantages of open nesting. Such a program semantics for open nesting would allow a user to be oblivious to open transactions in libraries. Unfortunately, our research has made us doubtful that program semantics can offer an elegant answer to the modularity question for open nesting.

Perhaps we should seek loopholes for TM other than open nesting. For example, Herlihy *et al.* [11] have proposed an early-release mechanism for dropping locations from a transaction's readset or writeset. Zilles and Baugh [27] have suggested a mechanism for pausing and resuming a transaction to allow the execution of non-transactional code. Harris [10] has proposed an external-action abstraction for performing I/O. We have not studied these models enough to say whether like open nesting, they provide anomalous or difficult semantics.

If ever a safe loophole can be punched in the steel armor of classical transaction memory, however, we believe that a precise understanding of the system's memory semantics will be necessary. We hope that our work will offer insight into how transactional-memory loopholes, such as open nesting, might be safely introduced.

ACKNOWLEDGEMENTS

We thank Eliot Moss of the University of Massachusetts and Scott Ananian, Bradley Kuszmaul, and Gideon Stupp of MIT CSAIL for helpful early discussions on the semantics of open nesting.

REFERENCES

[1] C. Beeri, P. A. Bernstein, and N. Goodman. A model for concurrency in nested transactions systems. *Journal of the ACM*, 36(2):230–269, 1989.

[2] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, Jun 2005.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition, 2001.

[4] C. Davies. Recovery semantics for a DB/DC system. In *ACM National Conference*, pages 136–141, 1973.

[5] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *SPAA*, pages 1–11, Newport, Rhode Island, June 1997.

[6] M. Frigo. The weakest reasonable memory model. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, Jan. 1998.

[7] M. Frigo and V. Luchangco. Computation-centric memory models. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*, pages 240–249, 1998.

[8] J. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Databases*, pages 144–154, Sept. 1981.

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[10] T. Harris. Exceptions and side-effects in atomic blocks. In *In PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[11] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Principles of Distributed Computing*, pages 92–101, New York, NY, USA, 2003. ACM Press.

[12] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *International Symposium on Computer Architecture*, 1993.

[13] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *POPL'81: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–218, New York, NY, USA, 1981. ACM Press.

[14] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[15] B. Liblit. An operational semantics for LogTM. Technical report, Department of Computer Sciences, University of Wisconsin-Madison, August 2006.

[16] A. McDonald, J. Chung, B. D. Carlstrom, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *International Symposium on Computer Architecture*, June 2006.

[17] J. E. B. Moss. Nested transactions and reliable distributed computing. In *SRDS*, pages 33–39, Pittsburgh, PA, July 1982.

[18] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, MA, USA, 1985.

[19] J. E. B. Moss. Open nested transactions:semantics and support. In *Workshop on Memory Performance Issues*, Austin, Texas, Feb 2006.

[20] J. E. B. Moss, N. D. Griffeth, and M. H. Graham. Abstraction in recovery management. In *SIGMOD*, pages 72–83, 1986.

[21] J. E. B. Moss and A. L. Hosking. Nested transactional memory: Model and preliminary architecture sketches. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL) Conference Proceedings*, San Diego, California, Oct. 2005.

[22] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[23] D. P. Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[24] M. L. Scott. Sequential specification of transactional memory semantics. In *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[25] I. Traiger. Trends in systems aspects of database management. In *International Conference on Databases*, pages 1–21. Wiley Heyden Ltd, 1983.

[26] G. Weikum and H.-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In *Database Transaction*

Models for Advanced Applications, pages 515–553. Morgan Kaufmann, San Francisco, CA, USA, 1992.

[27] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.