Modular Transactions: Bounding Mixed Races in Space and Time

Brijesh Dongol
University of Surrey, UK
b.dongol@surrey.ac.uk

Radha Jagadeesan
DePaul University, USA
rjagadeesan@cs.depaul.edu

James Riely
DePaul University, USA
jriely@cs.depaul.edu

Abstract

We define *local transactional race freedom* (LTRF), which provides a *programmer model* for *software transactional memory*. LTRF programs satisfy the *SC-LTRF* property, thus allowing the programmer to focus on sequential executions in which transactions execute atomically. Unlike previous results, *SC-LTRF* does not require global race freedom. We also provide a lower-level *implementation model* to reason about *quiescence fences* and validate numerous compiler optimizations.

CCS Concepts • **Theory of computation → Parallel computing models**; **Abstraction**;

1 Introduction

For concurrent programs communicating via a shared-memory subsystem that includes locks, the *SC-DRF* property states that a *Data Race Free* program can be fully understood by considering only executions that are *Sequentially Consistent*, meaning that the shared-memory subsystem can be modeled as a standard sequential store [3].

For programs that use transactions to augment or replace locking, the analogous SC-*T*RF property states that for *Transactionally Race-Free* programs, it suffices to consider executions that are SC and where transactions are executed atomically. For TRF programs, SC-TRF implies *opacity* [15, 16], which generalizes SC-DRF to include aborted and live transactions. SC-TRF is a conditional form of *operational refinement*: for TRF programs, "every behavior a user can observe of a program using a TM implementation can also be observed when the program uses an abstract TM that executes each block atomically" [22].

Dongol is supported by EPSRC grant EP/R032556/1. Jagadeesan and Riely are supported by National Science Foundation CCR-1617175.

PPoPP '19, February 16–20, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6225-2/19/02…$15.00
https://doi.org/10.1145/3293883.3295708

Reasoning with SC-TRF is powerful, particularly for *mixed-mode access*, where a single location is accessed both transactionally and nontransactionally. A common idiom is *privatization*, shown in the following program.

$$\mathsf{atomic}_a\ \{\ \mathsf{if}\ !y\ \mathsf{then}\ x := 1\ \}\ \|\ \mathsf{atomic}_b\ \{\ y := 1\ \};\ x := 2$$

Here, there are two threads, separated by parallel composition. Transactions are denoted by atomic blocks, with transaction names as subscripts to facilitate discussion. The first thread atomically reads y and updates x if y is 0 (the initial value). The second thread atomically writes y, then executes a plain (nontransactional) write to x.

Reasoning sequentially and assuming all transactions commit, it is impossible for the program to terminate with $x = 1$ since the atomic blocks must appear to occur in some serial order. Suppose a serializes first—then the write of 1 to x, denoted $\langle \mathrm{W}x1 \rangle$, must precede $\langle \mathrm{W}x2 \rangle$, and the final result is 2. Suppose b serializes first—then there will be no $\langle \mathrm{W}x1 \rangle$, since the only available value for y is 1.

Thus, the atomic blocks are used to synchronize threads. In the case that $x := 2$ is replaced with some costly computation, the privatization idiom can be used to reduce computational costs inside atomic blocks.

The reverse idiom is *publication*, exemplified by:

$$x := 1;\ \mathsf{atomic}_a\ \{\ y := 1\ \}\ \|\ \mathsf{atomic}_b\ \{\ z := 2;\ \mathsf{if}\ y\ \mathsf{then}\ z := x\ \}$$

Reasoning as before, it is impossible for the program to terminate with $z = 0$. Suppose transaction a serializes first—then b must see both $\langle \mathrm{W}x1 \rangle$ and $\langle \mathrm{W}y1 \rangle$ and therefore end by writing $\langle \mathrm{W}z1 \rangle$. Suppose b serializes first—then there will be no second write to z, since the only available value for y is the initial value 0, and thus the last write to z is $\langle \mathrm{W}z2 \rangle$.

It is a direct consequence of sequential reasoning that these outcomes must be forbidden. In the implementation of Software Transactional Memory (STM), many performance enhancements, such as optimistic execution, can result in a failure of SC-TRF, allowing behaviors such as those above. This has led to a tension between the programmer model and the implementation of STMs, resulting in a great literature on the subject, with many competing notions of *transactional race* that abstract away implementation details to a greater or lesser degree [1, 17, 24, 28].

In this paper, we emphasize the *programmer model*, developing a high-level definition of a transactional race that makes mixed-mode idioms safe by definition (§2 and §4). We attempt to make the programmer model as broadly applicable as possible by adapting the notion of *local* data race

developed by Dolan et al. [9]. At the same time, we show that our model is efficiently implementable, in that it avoids common pitfalls that overly constrain the STM implementation, such as *publication by antidependency* or *global lock atomicity* (§3). Our programmer model disables common compiler optimizations; so, we develop a slightly more concrete *implementation model* that supports compiler optimizations (§5). We describe how to compile our model to x86 and ARMv8 (§6). We are inspired by Khyzha et al. [22], who followed the same agenda for global races using a model similar to our implementation model; we discuss related work in §7.

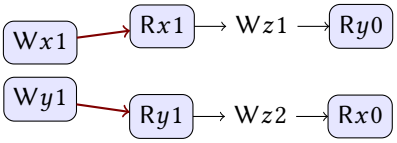
In addition to providing a novel programmer model, this paper extends existing work in several ways.

Local Race Freedom. The SC-TRF property is a *global* property: a race anywhere in the program is sufficient to nullify the TRF property, typically resulting in undefined semantics. Recently, Dolan et al. [9] proposed *local* DRF as an alternative to global DRF for programs that synchronize via Java-like volatiles. We propose the first *local TRF* property.

Local TRF is strictly more expressive than the global TRF models considered in prior work. As a result, we are able to provide an SC-*L*TRF guarantee, which applies to many additional programs. Consider the variant of the well-known *independent reads of independent writes* example below.

$$\begin{array}{l}\quad \text{atomic}\ \{\, x := 1 \,\} \parallel \text{atomic}\ \{\, y := 1 \,\} \\ \parallel \text{atomic}\ \{\, r_1 := x \,\};\ z := 1;\ \text{atomic}\ \{\, r_2 := y \,\} \\ \parallel \text{atomic}\ \{\, q_1 := y \,\};\ z := 2;\ \text{atomic}\ \{\, q_2 := x \,\}\end{array} \qquad \text{(IRIW)}$$

The following outcome cannot occur sequentially.



If the writes to z are removed, then SC-TRF reasoning allows a programmer to conclude that this sequence of reads cannot occur. However, with the writes to z included, SC-TRF reasoning says nothing about this program, since, by any definition, there is a race on z. SC-LTRF allows us to ignore this race. Since no transactional variable is involved in a race, we are guaranteed that every execution of this program behaves as though the *transactional portion* were executed sequentially with no interleaving of transactions. This example illustrates *spatial locality*.

To understand the temporal flavor of locality, consider the following program that uses IRIW as a parallel component.

$$\begin{array}{l}\quad x := -1;\ \text{atomic}\ \{\, F{+}{+} \,\} \parallel x := -2;\ \text{atomic}\ \{\, F{+}{+} \,\} \\ \parallel \text{atomic}\ \{\, r := F \,\};\ \text{if}\ r = 2\ \text{then IRIW}\end{array}$$

Again, standard SC-TRF reasoning says nothing about this program, since there are races on x. But there is no race on x or y *after* the guard $r = 2$ becomes true; SC-*L*TRF allows us to reason sequentially from that point, ensuring that IRIW behaves as expected.

Thus, by adapting the notion of locality from [9], we enable *modular reasoning with transactions* by isolating transactional races from other data races, in both space and time.

Defined Behavior for Racy Programs. Most prior models based on SC-TRF either give undefined semantics for programs with races or assume that the underlying memory model is sequentially consistent. We define the semantics of programs using the relaxed memory model of [9], and thus give a defined semantics for racy programs using a realistic memory model.

Implementation-Level Reasoning. Most prior work relies on programmers to place quiescence fences to guarantee safety [22, 27, 34–36]. We connect our high-level model to this previous work by developing a lower-level implementation model that includes explicit fences. Our lower-level assumes only that the underlying transactional machinery provides order between transactions that have a direct dependency, e.g., as in the publication idiom. We note that hardware transactions [5, 6, 10] support the ordering assumptions of our lower-level model. Fences are necessary only to provide order when there is no direct dependency, as in the privatization idiom. We provide a correctness criterion to realize our abstract programming model, and compare the fences required to realize our high-level model to previous approaches.

In addition to building on these aspects of prior work on SC-TRF, we prove that common compiler optimizations are sound under LTRF. In addition to *all* of the optimizations validated by LDRF [9], we also validate some optimizations specific to transactions, inspired by optimizations that are sound with respect to locks [27]. For example, we show that empty transactions can be elided, that the scope of transactions can be increased, and that adjacent transactions can be combined.

2 Programmer Model

Dolan et al. [9] give a semantics for a language using Java-like volatiles for synchronization. We adapt their semantics to *isolated* transactions [13, 26] (where plain actions may not be causally interleaved with transactional actions). Transactions are more general than volatiles in several ways:

- A transaction may abort.
- A transaction may both read and write.
- A transaction may access more than one location.
- The same location may be used in both transactional and plain accesses.

We give the semantics of a program as a set of *traces*, each of which is a sequence of *actions* (e.g., read, write, transaction begin). Dolan et al. [9] give both an operational semantics generating traces and an axiomatic semantics defined over event graphs. We concentrate on the axiomatic treatment, treating actions as events in an event graph and deriving orders over these actions. We use the words *trace* and *execution*

interchangeably, preferring "trace" when the exact sequence of actions is relevant and "execution" when it suffices to consider the derived relations.

We have designed the semantics so that transactions behave exactly like the volatiles of [9] for *degenerate* traces in which each transaction contains a single read or write action, transactional and nontransactional locations are disjoint, and each transaction is committed and contiguous.

In this section, we present a programmer model that validates mixed-mode idioms such as privatization, but fails to validate common compiler optimizations. In §5, we give a low-level model that validates compiler optimizations, but only conditionally validates mixed-mode idioms.

Actions The syntax of actions is as follows.

$a, b, c \in Act$	(Action Id)	α	$::= \langle a{:}s\mathrm{W}xvq\rangle$	(Write)
$s, t \in Thrd$	(Thread Id)		$\mid \langle a{:}s\mathrm{R}xvq\rangle$	(Read)
$x, y \in Loc$	(Location)		$\mid \langle a{:}s\mathrm{B}\rangle$	(Begin)
$v, w \in \mathbb{Z}$	(Value)		$\mid \langle a{:}s\mathrm{C}b\rangle$	(Commit)
$q, p \in \mathbb{Q}$	(Timestamp)		$\mid \langle a{:}s\mathrm{A}b\rangle$	(Abort)

Action ids are unique identifiers for actions. Thread ids include the reserved thread id init, used for initialization. To simplify the definition of initialization, we assume that the set of locations is finite. We take values to be integers and timestamps to be rationals, as in [9].

The write action $\langle a{:}s\mathrm{W}xvq\rangle$ denotes a write of v to x by thread s, with action name a. Likewise, $\langle a{:}s\mathrm{R}xvq\rangle$ denotes a read. The timestamp q is used to encode relations between these actions, as detailed below.

The begin action $\langle b{:}s\mathrm{B}\rangle$ denotes the begin of transaction by thread s, with action name b. We also use b as the transaction name. The commit action $\langle a{:}s\mathrm{C}b\rangle$ denotes the commit of the transaction named b. Likewise $\langle a{:}s\mathrm{A}b\rangle$ denotes the abort of b. We refer to commits and aborts collectively as *resolution* actions.

We often drop components of the action syntax that are not interesting for the discussion at hand, e.g., we may write $\langle a{:}s\mathrm{W}xvq\rangle$ as either $\langle a\rangle$, $\langle a{:}s\rangle$, $\langle \mathrm{W}x\rangle$, $\langle \mathrm{W}xv\rangle$, or $\langle \mathrm{W}xq\rangle$.

Traces and Transactions. A *trace* is a finite sequence of actions $\alpha_1\alpha_2\cdots\alpha_n$. We use σ, ρ to range over traces. We only consider *well-formed* traces (defined below), which begin with an *initializing transaction* of the form $\langle b{:}\mathrm{init}\,\mathrm{B}\rangle\ \langle \mathrm{init}\,\mathrm{W}x_1v_1 0\rangle\cdots\langle \mathrm{init}\,\mathrm{W}x_nv_n 0\rangle\langle \mathrm{init}\,\mathrm{C}b\rangle$, which contains exactly one write for each location, at timestamp 0. Here init is a reserved thread name. In examples, we usually omit this initializing transaction, assuming that all locations are initialized to 0.

Each trace $\sigma = \alpha_1\alpha_2\cdots\alpha_n$ generates a total order $\xrightarrow{\text{index}}_\sigma$, where $\alpha_i \xrightarrow{\text{index}}_\sigma \alpha_j$ iff $i < j$. Usually, the trace is clear from context and we drop the subscript, preferring $\xrightarrow{\text{index}}$ to $\xrightarrow{\text{index}}_\sigma$. We adopt this convention throughout, dropping the subscript in definitions as well as examples.

We derive several other relations from a trace, including *initialization order*, *program order*, *write-to-write order* (aka *coherence*) and *write-to-read order* (aka *reads-from*).

- $\langle a{:}s\rangle \xrightarrow{\text{init}} \langle b{:}t\rangle$ iff $s = \mathrm{init} \neq t$.
- $\langle a{:}s\rangle \xrightarrow{\text{po}} \langle b{:}t\rangle$ iff $a \xrightarrow{\text{index}} b$ and $s = t$.
- $\langle a{:}\mathrm{W}xq\rangle \overset{\text{ww}}{\dashrightarrow} \langle b{:}\mathrm{W}yp\rangle$ iff $x = y$ and $q < p$.
- $\langle a{:}\mathrm{W}xvq\rangle \xrightarrow{\text{wr}} \langle b{:}\mathrm{R}ywp\rangle$ iff $x = y$, $v = w$ and $q = p$.

All of these relations are irreflexive. $\xrightarrow{\text{po}}$ and $\overset{\text{ww}}{\dashrightarrow}$ are transitive. The domain and range are disjoint for $\xrightarrow{\text{init}}$ and $\xrightarrow{\text{wr}}$.

In the context of a trace, we often refer to actions by name. For example, we prefer "$a \xrightarrow{\text{po}} b$" to "$\langle a\rangle \xrightarrow{\text{po}} \langle b\rangle$". We also write "$a = \langle s\mathrm{W}xvq\rangle$" rather than "$\exists i.\ \alpha_i = \langle a{:}s\mathrm{W}xvq\rangle$".

We take the name of the begin action to be the unique id for each transaction. We say that action a *belongs to* transaction b if $\langle b{:}\mathrm{B}\rangle \xrightarrow{\text{po}} a$ and there is no commit or abort action c such that $b \xrightarrow{\text{po}} c \xrightarrow{\text{po}} a$. We say that a is *transactional* if it belongs to some transaction, and *plain* otherwise.

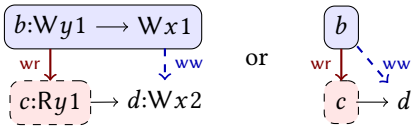
Each trace induces an equivalence over action names, relating actions that belong to the same transaction:

$a \overset{\text{tx}}{\sim} b$ iff $a = b$ or a and b belong to the same transaction.

Note that plain actions are included in $\overset{\text{tx}}{\sim}$, although they only relate to themselves.

There are three possible states for transactions: *committed*, *aborted* and *live*. Committed and aborted transactions are *resolved*. Committed and live transactions are *nonaborted*. We use the same terminology to refer all of the actions in a transaction; thus, we may use "aborted write action" to refer to a write action that belongs to an aborted transaction.

We visualize traces as graphs. For example, the trace $\langle a{:}\mathrm{init}\,\mathrm{B}\rangle\ \langle \mathrm{init}\,\mathrm{W}x00\rangle\ \langle \mathrm{init}\,\mathrm{W}y00\rangle\ \langle \mathrm{init}\,\mathrm{C}a\rangle\ \langle b{:}s\mathrm{B}\rangle\ \langle s\mathrm{W}y11\rangle\ \langle s\mathrm{W}x11\rangle\ \langle s\mathrm{C}b\rangle\ \langle c{:}t\mathrm{B}\rangle\ \langle t\mathrm{R}y11\rangle\ \langle t\mathrm{A}c\rangle\ \langle d{:}t\mathrm{W}x22\rangle$ is visualized as:



To avoid clutter, we drop the label on $\xrightarrow{\text{po}}$ and elide the initializing transaction. Instead of including explicit begin and resolution actions, we visualize transactions using boxes. Committed and live transactions are drawn in solid boxes, colored blue. Aborted transactions are drawn in dashed boxes, colored red.

Well-Formedness. A trace is a *well-formed* if each of the following hold:

- WF_1. The trace starts with an initializing transaction.
- WF_2. Action names are unique: if $a \xrightarrow{\text{index}} b$, then $a \neq b$.
- WF_3. Write timestamps are per-location unique: If $a = \langle \mathrm{W}xq\rangle$ and $b = \langle \mathrm{W}xq\rangle$, then $a = b$.
- WF_4. Each begin action has at most one resolution, and each resolution has exactly one begin action.
- WF_5. Each resolution follows its begin in $\xrightarrow{\text{po}}$, without an intervening begin or resolution.

WF_6. If b is a read, then there is some a such that $a \xrightarrow{wr} b$.
WF_7. If $a \xrightarrow{wr} b$ and a is aborted or live, then $a \overset{tx}{\simeq} b$.
WF_8. If $a \xrightarrow{wr} b$, then $a \xrightarrow{index} b$.
WF_9. If b is transactional, then there is no committed or live $c \xrightarrow{index} b$ such that $b \dashrightarrow[ww] c$.
WF_{10}. If b is transactional and there is some transactional $a \xrightarrow{wr} b$, then there is no committed or live $c \xrightarrow{index} b$ such that $a \dashrightarrow[ww] c$.
WF_{11}. If b is transactional and there is some $a \xrightarrow{wr} b$, then there is no $c \overset{tx}{\simeq} b$ such that $c \xrightarrow{index} b$ and $a \dashrightarrow[ww] c$.

WF_1 ensures that locations are initialized. WF_2–WF_3 ensure that action names and timestamps are unique. WF_4–WF_5 ensure proper bracketing for transactions. These conditions also preclude nesting of transactions — we leave the treatment of nested transactions to future work. WF_6 ensures that all reads are fulfilled. WF_7 ensures that aborted and live writes are not visible outside the transaction.

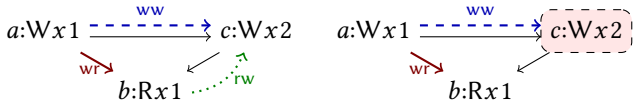
WF_8–WF_{11} constrain the interleavings allowed in a trace. For the most part, we view traces as abstract execution graphs, where transactions are expressed as multiple $\xrightarrow{po}$-contiguous actions. In execution graphs, time is *relative*: it is expressed as the *happens-before* relation, which captures causal relations between actions. At the concrete level of a trace, time is *absolute*: it is expressed by order in the sequence. Viewed as execution graphs, WF_8–WF_{11} are redundant with respect to consistency criteria given below. These conditions, instead, constrain the concrete representation of the execution graph as a trace, enabling inductive reasoning that mirrors the operational reasoning of [9].

WF_8 ensures that reads only see the absolute past: reads are not allowed to "see the future". This condition is guaranteed by the operational semantics of [9], but here must be stated explicitly. There is no similar requirement that writes respect absolute time. They may appear out of order. For example, we allow the trace $\langle Wx22\rangle\langle Wx11\rangle$.

WF_9–WF_{11} constrain the interleaving of the actions from different transactions. There is no analogue of these rules in [9] since volatiles are expressed as a single action. WF_9 forbids $\langle cWx22\rangle\langle bWx11\rangle$ when both are transactional — we ignore aborted writes because they are not visible to other transactions. WF_{10} forbids $\langle aWx11\rangle\langle cWx22\rangle\langle bRx11\rangle$ when all three are transactional. WF_{11} forbids $\langle aWx11\rangle\langle cWx22\rangle\langle bRx11\rangle$ when $c \overset{tx}{\simeq} b$.

Antidependencies. An *antidependency* relates a read to any write that cannot precede it. We use $\overset{rw}{\dashrightarrow}$ to represent antidependency as *read-to-write order* (aka *from-read*). Ignoring transactions, $b \overset{rw}{\dashrightarrow} c$ whenever $a \xrightarrow{wr} b$ and $a \dashrightarrow[ww] c$, for some a.

As we shall see, antidependencies are not allowed to contradict the happens-before order, which defines causality. The end result is that stale reads are precluded. For example, consider the trace $\langle a{:}sWx1\rangle\langle c{:}sWx2\rangle\langle b{:}sRx1\rangle$. This trace should not be allowed, since it reads 1 after writing 1 and then 2 in the same thread. Because $c \xrightarrow{po} b \overset{rw}{\dashrightarrow} c$, this trace, shown on the left below, will not be considered consistent:



Aborted transactions complicate the definition of antidependency. For example, if c is part of an aborted transaction, as shown on the right, then the outcome should be allowed. Note that if b and c belonged to the same aborted transaction, then the execution would be disallowed by condition WF_{11} in the definition of well-formed trace.

Thus we arrive at the following definition:

> $b \overset{rw}{\dashrightarrow} c$ iff $a \xrightarrow{wr} b$ and $a \dashrightarrow[ww] c$, for some a, and c is either plain or nonaborted.

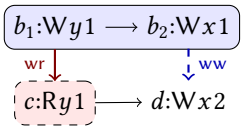
Lifted Relations. A common technique to enforce transactional atomicity is to *lift* orders from individual actions to the level of transactions [6, 10, 32]. Notationally, we indicate a lifted relation by prefixing "l." For example, the lifting of $\xrightarrow{wr}$ is written $\xrightarrow{lwr}$. We also use two variants.

- $\xrightarrow{lR}$ is the lifting of relation $\xrightarrow{R}$.
- $\xrightarrow{xR}$ restricts $\xrightarrow{lR}$ to transactions.
- $\xrightarrow{cR}$ restricts $\xrightarrow{xR}$ to nonaborted transactions.

For any relation $\xrightarrow{R}$, the definitions are as follows.

- $a \xrightarrow{lR} b$ iff $a \xrightarrow{R} b$ or $a' \xrightarrow{R} b'$ for some $a' \overset{tx}{\simeq} a \not\overset{tx}{\simeq} b \overset{tx}{\simeq} b'$.
- $a \xrightarrow{xR} b$ iff $a \xrightarrow{lR} b$ and a, b are transactional.
- $a \xrightarrow{cR} b$ iff $a \xrightarrow{xR} b$ and a, b are committed or live.

Consider the following execution, where we label the individual actions of b.



We have $b_1 \xrightarrow{wr} c$ but not $b_2 \xrightarrow{wr} c$. In the lifted relation both of these hold; in particular, we have $b_2 \xrightarrow{lwr} c$. Similarly, we have $b_1 \dashrightarrow[lww] d$ but not $b_1 \dashrightarrow[ww] d$. The "x" variants exclude d. The "c" variants exclude both c and d.

Summarizing the relations defined thus far, we have:

- $\xrightarrow{index}$ is the absolute order of events in a trace.
- $\xrightarrow{init}$ relates initialization events to other events.
- $\xrightarrow{po}$ restricts $\xrightarrow{index}$ to events of same thread.
- $\dashrightarrow[ww]$ is write-to-write order, derived from timestamps.
- $\xrightarrow{wr}$ is write-to-read order, derived from timestamps.
- $\overset{rw}{\dashrightarrow}$ is read-to-write order, derived from $\dashrightarrow[ww]$ and $\xrightarrow{wr}$.

Lifting is only applied to the last three relations.

Happens-Before. The *happens-before* order, $\xrightarrow{hb}$, is a partial order that captures dependency, or causality, between actions. It serves a crucial role in understanding distributed systems. In the next subsection, happens-before is used to define *consistent* executions that obey the intended notion of causality. In §4, happens-before is also used to define

data races. By varying the definition of happens-before, we change the definition of both consistency and raciness.

We define $\xrightarrow{\text{hb}}$ to be the least relation that is closed with respect to the following.

$$a \xrightarrow{\text{hb}} c \text{ if } a \,(\xrightarrow{\text{init}} \cup \xrightarrow{\text{po}} \cup \xrightarrow{\text{cwr}} \cup \dashrightarrow^{\text{cww}})\, c \qquad (\text{HB}_{\text{BASE}})$$
$$a \xrightarrow{\text{hb}} c \text{ if } a \xrightarrow{\text{hb}} b \xrightarrow{\text{hb}} c \qquad (\text{HB}_{\text{TRANS}})$$
$$a \xrightarrow{\text{hb}} c \text{ if } c \text{ is plain, } a \dashrightarrow^{\text{lww}} c \text{ and } a \overset{\text{crw}}{\cdots\!\rightarrow} b \xrightarrow{\text{hb}} c \qquad (\text{HB}_{\text{WW}})$$

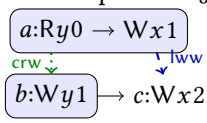
We discuss variations of HB_{WW} at the end of this section. We discuss an alternative model without HB_{WW} in §5.

By HB_{BASE}, happens-before includes initialization order, program order, lifted write-to-write order and lifted write-to-read order. HB_{TRANS} says that happens-before is transitive. These rules are adapted from the analogous rules in [9]. The only subtlety of these rules lies in the choice of lifted relation in HB_{BASE}; note that we restrict HB_{BASE} to include order only from committed and live transactions. We discuss the reason for this in the next subsection.

HB_{WW} is designed to ensure that *privatization* is considered *race-free*. Roughly, two actions are *racing* if they touch a common location, neither is aborted, one is a write, and they are not ordered by $\xrightarrow{\text{hb}}$. HB_{WW} only applies when a and b are live or committed. If c is also live or committed, then this rule adds nothing: HB_{BASE} already gives us $a \xrightarrow{\text{hb}} c$ since $a \dashrightarrow^{\text{cww}} c$.

Example 2.1. Recall the *privatization* example from §1.

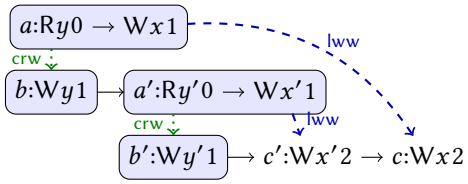
$\text{atomic}_a\,\{\text{ if } !y \text{ then } x := 1\,\}$
$\parallel \text{atomic}_b\,\{\,y := 1\,\};\, x := 2$



Without HB_{WW}, there would be a race between $\langle \text{W}x1\rangle$ and $\langle \text{W}x2\rangle$. By including $a \dashrightarrow^{\text{lww}} c$ in happens-before, we ensure that this execution is considered race free.

Order from HB_{WW} can cascade, as in the following.

$\text{atomic}_a\,\{\text{ if } !y \text{ then } x := 1\,\}$
$\parallel \text{atomic}_b\,\{\,y := 1\,\};\, \text{atomic}_{a'}\,\{\text{ if } !y' \text{ then } x' := 1\,\}$
$\parallel \text{atomic}_{b'}\,\{\,y' := 1\,\};\, x' := 2;\, x := 2$



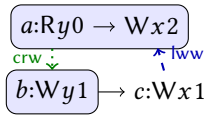
Consistency. We say that an execution is *consistent* iff it is well-formed and the following hold.

$$(\xrightarrow{\text{hb}} \cup \xrightarrow{\text{lwr}} \cup \overset{\text{xrw}}{\cdots\!\rightarrow})\text{ is acyclic.} \qquad (\text{Causality})$$
$$(\xrightarrow{\text{hb}}; \dashrightarrow^{\text{lww}})\text{ is irreflexive.} \qquad (\text{Coherence})$$
$$(\xrightarrow{\text{hb}}; \overset{\text{lrw}}{\cdots\!\rightarrow})\text{ is irreflexive.} \qquad (\text{Observation})$$
$$(\overset{\text{crw}}{\cdots\!\rightarrow}; \xrightarrow{\text{hb}}; \dashrightarrow^{\text{lww}})\text{ is irreflexive.} \qquad (\text{Anti}_{\text{WW}})$$

Causality, Coherence and Observation all appear in [9]. We discuss Anti_{WW} below and in Example 3.5.

Example 2.2. Consider the variant of Example 2.1, in which the writes on x are given the reverse order in $\dashrightarrow^{\text{lww}}$.

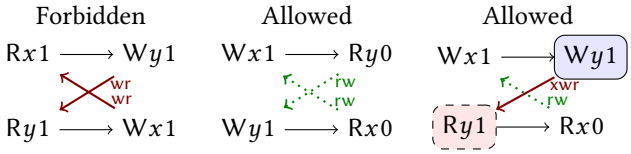
$\text{atomic}_a\,\{\text{ if } !y \text{ then } x := 2\,\}$
$\parallel \text{atomic}_b\,\{\,y := 1\,\};\, x := 1$



Intuitively, this execution should be disallowed since $\dashrightarrow^{\text{lww}}$ seems to order the writes incorrectly. Anti_{WW} forbids it.

Technically, this execution must be disallowed in order to establish the SC-LTRF theorem, which states that any race can be discovered in a sequential execution. To see the issue, note that the two writes on x are not ordered by $\xrightarrow{\text{hb}}$ (HB_{WW} does not apply here); thus they are in a race. SC-LTRF requires, therefore, that we find a sequential execution of this program that also exhibits a race. But this is impossible: any sequential execution must have a before b, and therefore before c, and thus $a \dashrightarrow^{\text{lww}} c$. But in this case, HB_{WW} adds order between a and c, eliminating the race.

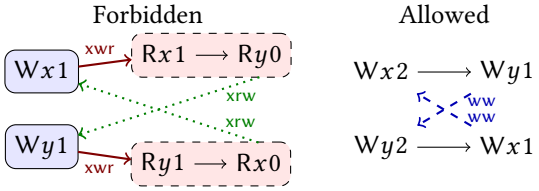
As noted in [9], since $\xrightarrow{\text{po}} \subseteq \xrightarrow{\text{hb}}$ and $\xrightarrow{\text{wr}} \subseteq \xrightarrow{\text{lwr}}$, the inclusion of $\xrightarrow{\text{lwr}}$ in Causality forbids "load buffering," shown on the left below, which is allowed by many other models.



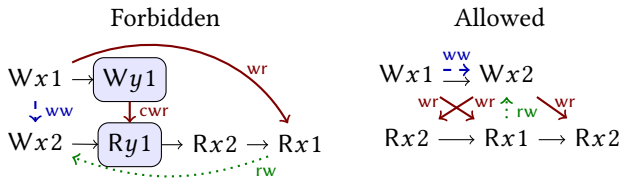
On the other hand, the model does allow "store buffering," shown in the middle above, since plain antidependencies only have an irreflexivity requirement in Observation, not an acyclicity requirement.

We do not include aborted transactions in HB_{BASE}; in conjunction, with Observation, this would cause publication through aborted reads. To see this, consider the execution on the right above, which is allowed by our model, but would be disallowed if $\xrightarrow{\text{hb}}$ included $\xrightarrow{\text{xwr}}$ rather than $\xrightarrow{\text{cwr}}$.

Were we to use $\overset{\text{crw}}{\cdots\!\rightarrow}$ in Causality, the execution on the left below would be allowed. But this execution violates opacity, which requires a total order among all transactions (including aborted transactions) that is consistent with happens-before order [15, 16]. Therefore the execution must be forbidden. If the writes are plain, however, this execution is similar to the store buffering example, and should be allowed. Thus, it would be too strong to use $\overset{\text{lrw}}{\cdots\!\rightarrow}$ in Causality, or to require acyclicity of $(\xrightarrow{\text{hb}} \cup \overset{\text{lrw}}{\cdots\!\rightarrow})$ in Observation. Similarly, we cannot use $\dashrightarrow^{\text{lww}}$ in Causality or require acyclicity of $(\xrightarrow{\text{hb}} \cup \dashrightarrow^{\text{lww}})$ in Coherence. In either case, we would rule out the execution on the right.



As noted in [9], the notion of coherence in LTRF is stronger than Java, which allows the execution on the left below. On the other hand, LTRF coherence is not as strong as coherence in hardware models and C++ atomics, which forbid the execution on the right—allowing such executions is necessary to support compiler optimizations, such as common subexpression elimination [9, 31].



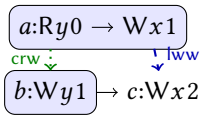
Anti-Dependence vs Happens-Before. HB_{ww} adds to $\xrightarrow{\text{hb}}$ the minimal order needed to validate privatization. There is a design space of choices for additional constraints that can be imposed on the compositions of $\cdots\!\!\xrightarrow{\text{crw}}$ and $\xrightarrow{\text{hb}}$.

Example 2.3. There are six variants, each of which we illustrate with an example. For completeness, we include HB_{ww} with a variant of Example 2.1. Following Example 2.2, many of these require an additional antidependency axiom. The exceptions involve $\xrightarrow{\text{lwr}}$, for which Causality suffices.

$a \xrightarrow{\text{hb}} c$ if c is plain, $a \dashrightarrow_{\text{lww}} c$ and $a \cdots\!\!\xrightarrow{\text{crw}} b \xrightarrow{\text{hb}} c$ (HB_{ww})

$(\cdots\!\!\xrightarrow{\text{crw}}; \xrightarrow{\text{hb}}; \dashrightarrow_{\text{lww}})$ is irreflexive. $(\text{Anti}_{\text{ww}})$

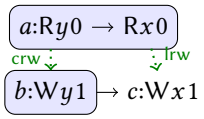
$\text{atomic}_a\{r := y; x := 1\}$ ‖ $\text{atomic}_b\{y := 1\}; x := 2$



$a \xrightarrow{\text{hb}} c$ if c is plain, $a \cdots\!\!\xrightarrow{\text{lrw}} c$ and $a \cdots\!\!\xrightarrow{\text{crw}} b \xrightarrow{\text{hb}} c$ (HB_{rw})

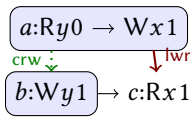
$(\cdots\!\!\xrightarrow{\text{crw}}; \xrightarrow{\text{hb}}; \cdots\!\!\xrightarrow{\text{lrw}})$ is irreflexive $(\text{Anti}_{\text{rw}})$

$\text{atomic}_a\{r := y; q := x\}$ ‖ $\text{atomic}_b\{y := 1\}; x := 1$



$a \xrightarrow{\text{hb}} c$ if c is plain, $a \xrightarrow{\text{lwr}} c$ and $a \cdots\!\!\xrightarrow{\text{crw}} b \xrightarrow{\text{hb}} c$ (HB_{wr})

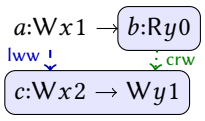
$\text{atomic}_a\{r := y; x := 1\}$ ‖ $\text{atomic}_b\{y := 1\}; q := x$



$a \xrightarrow{\text{hb}} c$ if a is plain, $a \dashrightarrow_{\text{lww}} c$ and $a \xrightarrow{\text{hb}} b \cdots\!\!\xrightarrow{\text{crw}} c$ (HB'_{ww})

$(\xrightarrow{\text{hb}}; \cdots\!\!\xrightarrow{\text{crw}}; \dashrightarrow_{\text{lww}})$ is irreflexive. $(\text{Anti}'_{\text{ww}})$

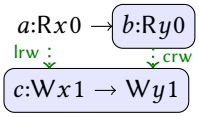
$x := 1; \text{atomic}_b\{r := y\}$ ‖ $\text{atomic}_c\{x := 2; y := 1\}$



$a \xrightarrow{\text{hb}} c$ if a is plain, $a \cdots\!\!\xrightarrow{\text{lrw}} c$ and $a \xrightarrow{\text{hb}} b \cdots\!\!\xrightarrow{\text{crw}} c$ (HB'_{rw})

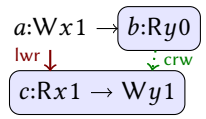
$(\xrightarrow{\text{hb}}; \cdots\!\!\xrightarrow{\text{crw}}; \cdots\!\!\xrightarrow{\text{lrw}})$ is irreflexive. $(\text{Anti}'_{\text{rw}})$

$q := x; \text{atomic}_b\{r := y\}$ ‖ $\text{atomic}_c\{x := 1; y := 1\}$



$a \xrightarrow{\text{hb}} c$ if a is plain, $a \xrightarrow{\text{lwr}} c$ and $a \xrightarrow{\text{hb}} b \cdots\!\!\xrightarrow{\text{crw}} c$ (HB'_{wr})

$x := 1; \text{atomic}_b\{r := y\}$ ‖ $\text{atomic}_c\{q := x; y := 1\}$

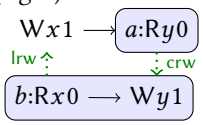


3 STM Design

We consider several examples from the literature to argue that the ordering required by our model does not impair efficient implementations of Software Transactional Memory.

Example 3.1. In accordance with [27, Figure 12], our model does not enforce *publication by antidependence*: The final outcome $r = q = 0$ is permitted in the program (left), as shown by the allowable execution (right).

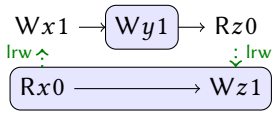
$x := 1; \text{atomic}_a\{r := y\}$ ‖ $\text{atomic}_b\{q := x; y := 1\}$



Note that if $\xrightarrow{\text{hb}}$ were to include $\cdots\!\!\xrightarrow{\text{crw}}$, then this execution would be forbidden by Observation. Note also that this execution is forbidden by any model that enforces Anti'_{rw}, from Example 2.3.

Example 3.2. In accordance with [27, Figure 11], our model does not enforce *global lock atomicity*: The final outcome $r = q = 0$ is possible in the program below.

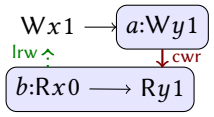
$x := 1; \text{atomic}_a\{y := 1\}; r := z$ ‖ $\text{atomic}_b\{q := x; z := 1\}$



This execution is allowed by all variants discussed in Example 2.3, including Anti'_{rw}.

Example 3.3. We now consider the limitations of our approach. Menon et al. [27] describes an idiom for *benign racy publication*. This outcome is considered desirable, yet our model forbids it: The final outcome $q = 0$ is *not* possible for the following program.

$x := 1; \text{atomic}_a\{y := 1\}$ ‖ $q := 2; \text{atomic}_b\{r := x;$ if y then $q := r\}$



The outcome is only allowed if b reads 0 for x and 1 for y, but this execution is disallowed by Observation.

Note that, in accordance with the name, the program is not race-free: the execution in which b reads 0 for y has a race on x. Thus, there is no canonical answer as to whether this execution is indeed benign and should be allowed.

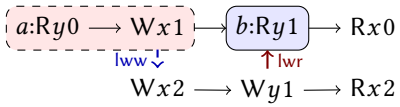
Example 3.4. The literature describes a class of STMs that implement *eager versioning*, which create an undo log for each write, perform writes as they are encountered (as opposed to during commits). If the transaction aborts, the updates are rolled back to their original logged values. Shpeisman et al. [34] describe potential issues with eager versioning in a mixed mode SC setting. In our relaxed memory setting, we show that these have natural explanations.

Consider the following program.

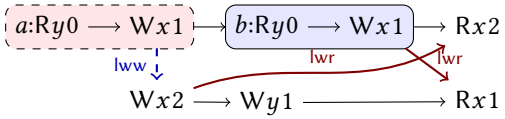
$$\begin{array}{l}\quad \text{atomic}_a\,\{\,\text{if } !y \text{ then } x := 1;\ \text{abort}\,\};\\ \quad \text{atomic}_b\,\{\,\text{if } !y \text{ then } x := 1\,\};\ r := x\\ \|\ x := 2;\ y := 1;\ q := x\end{array}$$

Under SC, the final value r=0 is considered to be problematic [34, Figure 3a] since it follows from a scenario in which the non-transactional write $\langle \mathrm{W}x2\rangle$ is lost, known as a *speculative lost update*. Assuming SC, suppose transaction a executes its write to x, then second thread executes its first two writes. Since transaction a aborts, the write to x would be rolled back to 0. Transaction b would then skip over the update to x (because it now observes $y = 1$). This allows $r = q = 0$.

In our setting, the final value $q = 0$ is immediately disallowed by $\mathrm{HB}_{\mathrm{BASE}}$ and Causality. Moreover, the first thread may read either 0 or 2 for x, whereas the second thread must read 2 for x, i.e., non-transactional write $\langle \mathrm{W}x2\rangle$ is not lost.



The scenario above may also result in executions such as:



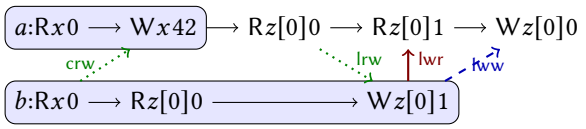
where transaction a successfully writes $\langle \mathrm{W}x1\rangle$. Again, the non-transactional write $\langle \mathrm{W}x2\rangle$ is available for the final reads in both threads.

Example 3.5. Analogous to eager versioning is a class of STMs that implement *lazy versioning* that cache writes locally within a transaction and update shared memory during a transaction's commit operation. Shpeisman et al. [34] discuss potential problems with lazy versioning in a mixed-mode setting. We consider the most interesting of these below.

Suppose z is an array in the program below.

$$\begin{array}{l}\quad \text{atomic}_a\,\{\,r := x;\ x := 42\,\};\ r_1 := z[r];\ r_2 := z[r];\ z[r] := 0\\ \|\ \text{atomic}_b\,\{\,q := x;\ \text{if } q \neq 42 \text{ then } z[q] := z[q] + 1\,\}\end{array}$$

The first thread atomically caches x and privatizes it by setting it to a special value (denoted here by 42). From a programmer's perspective $z[r]$ should not be read by other threads. However, in a lazy-versioning STM, transaction b may have been serialized before transaction a, yet contain a *buffered write* to $z[q]$. Thus, the reads of $z[r]$ may race with the buffered write to $z[q]$. A consequence of this is the execution below, where the two reads of $z[0]$ return different values.



The final outcome $r_1 \neq r_2$ is considered problematic in [34]. This outcome is disallowed by any variant of our model that includes $\textsc{Anti}_{\mathrm{RW}}$ (Example 2.3).

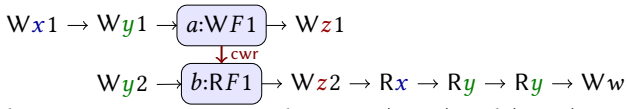
By $\textsc{Anti}_{\mathrm{WW}}$, the execution becomes inconsistent if we reverse the $\overset{\mathrm{lww}}{\dashrightarrow}$ order above. Thus, the outcome $z[0] \neq 0$ is forbidden by our model. This outcome is also considered problematic in [34].

4 Local Transactional Race Freedom

We introduce the concepts behind localising data race freedom (LDRF [9]) by example. Consider the program:

$$\begin{array}{l}\quad x := 1;\ y := 1;\ \text{atomic}_a\,\{\,F := 1\,\};\ z := 1\\ \|\ y := 2;\ \text{atomic}_b\,\{\,r := F\,\};\ z := 2;\ \text{if } r \text{ then } w := x + y - y\end{array}$$

Consider the case where b reads F from a, as depicted below. We leave the write-to-write orders and the values of the last four actions of the second thread unspecified.



There are write-write races between $\langle \mathrm{W}y1\rangle$ and $\langle \mathrm{W}y2\rangle$, and between $\langle \mathrm{W}z1\rangle$ and $\langle \mathrm{W}z2\rangle$. By some definitions of race, the write $\langle \mathrm{W}y1\rangle$ is also racing with the two reads of y. Thus, a global notion of race-freedom does not allow one to conclude anything about this program. A localised notion, however, would allow one to deduce that $\langle \mathrm{W}x1\rangle$ is correctly published to the second thread. Moreover, the two reads of y must see the same value and hence, the value written to w must be 1.

LDRF is defined relative to (1) a set Σ of traces, generated by the semantics of a program, (2) a set L of locations, and (3) a trace $\sigma \in \Sigma$, denoting a partial execution. For the example, Σ is fixed by the program. Let $L = \{x, y, F\}$. A race is an *L-race* if it involves a location in L; thus the race between $\langle \mathrm{W}z1\rangle$ and $\langle \mathrm{W}z2\rangle$ is not considered an L-race.

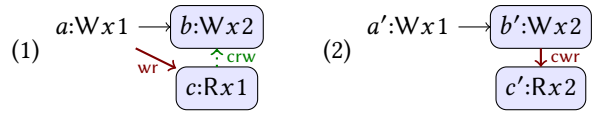
Now consider the trace $\sigma = \langle \mathrm{W}x1\rangle\langle \mathrm{W}y1\rangle\langle a{:}\mathrm{B}\rangle\langle \mathrm{W}F1\rangle \langle \mathrm{C}a\rangle\langle \mathrm{W}y2\rangle\langle b{:}\mathrm{B}\rangle\langle \mathrm{R}F1\rangle\langle \mathrm{C}b\rangle$ that linearizes the execution above. This σ contains an L-race between $\langle \mathrm{W}y1\rangle$ and $\langle \mathrm{W}y2\rangle$. Nonetheless, σ is *L-stable* for Σ because there is no $\sigma\rho \in \Sigma$ that includes an L-race between any action of σ and an action of ρ. It is important to note the definition of stability is relative to the set Σ. Trace σ is stable *for this program*, but would not be stable if, for example, the program is modified so that the first thread reads y after writing $z := 1$.

Having fixed σ, we now consider the *L-sequential* extensions of this prefix. These extensions are constrained to obey the sequential semantics for locations in L. Extensions that do not touch L, such as the writes to z, are unconstrained.

The SC-LDRF theorem says that either every extension of σ is L-sequential, or there is some L-sequential extension with an L-race. Since no L-sequential extension has a race, the program must behave sequentially from σ, guaranteeing that the read of x sees 1, that the two reads of y see the same value, and thus that the value written for w is 1.

The use of L in the definitions serves as an obvious spatial bound on races. The temporal bounds are less direct: By semantic fiat, future races can be ignored, since reads cannot see the future. By L-stability, past races are also excluded.

***From* D *to* T.** Locations used to store data are often disjoint from locations used to perform synchronization. In *T*RF, a single location may serve both purposes. This is the chief difficulty in extending L*D*RF to L*T*RF. Consider the program $x := 1;\ \text{atomic}\{x := 2\} \parallel \text{atomic}\{r := x\}$ with executions:



Since $\xrightarrow{\text{wr}}$ only creates happens-before order between committed transactions, there is a race in execution (1) but not (2). Consider the linearizations in which the read occurs last in the trace. We analyze by setting $L = \{x\}$. In trace abc, c is not L-sequential, whereas in $a'b'c'$, c' is L-sequential. In the SC-DRF theorem of [9], it is required that whenever there is a nonsequential racy read at the end of trace, such as c, we must be able to find a trace with a sequential read, such as c', that preserves the race. But here, this is impossible.

Note, however, that ac is L-sequential and has an L-race. In generalizing the SC-DRF theorem of [9] to mixed accesses, we must consider such prefixes. When transactional and plain accesses are disjoint this is not necessary, since well-formedness already guarantees sequential order between transactions. But well-formedness does not constrain interactions between transactional and plain access.

Intuitively, [9] proves that data races can be discovered by *sequential reasoning*. In the case of transactions, this is not enough. We must also have that all data races can be discovered by *executing transactions one-at-time*. To achieve this, we generalize the theorem to allow *permutations* that *preserve order* while ensuring that all actions of a transaction are *contiguous* in the trace.

***L*-Races.** Two actions are in *L-conflict* if they both access the same $x \in L$, at least one is plain, at least one is a write, and neither is aborted.

We say that (b, c) is an *L-race* if b and c are in L-conflict and $b \xrightarrow{\text{index}} c$, but not $b \xrightarrow{\text{hb}} c$. Two transactional actions cannot be in a race.

In global DRF, conflicting actions must be ordered by $\xrightarrow{\text{hb}}$; local DRF additionally constrains the direction of the order. This captures one form of temporal locality: future actions cannot causally affect the past.

***L*-Sequentiality and *L*-Stability.** For $L \subseteq Loc$, we say that c is *L-sequential* if c does not touch any location in L, or if c is a B, C, or A action, or if we have both of the following:

1. there is no $b \xrightarrow{\text{index}} c$ such that $c \dashrightarrow^{\text{ww}} b$, and
2. if $a \xrightarrow{\text{wr}} c$ then there is no $b \xrightarrow{\text{index}} c$ such that $a \dashrightarrow^{\text{ww}} b$.

Condition (1) applies when c is a write; it ensures that the timestamp chosen for c is larger than all preceding timestamps. Condition (2) applies when c is a read; it ensures that c reads from the preceding write with the largest timestamp.

An action that is not L-sequential is *L-weak*. Any L-weak action participates in an L-race: for writes, this follows from Coherence; for reads, from Observation.

Let Σ be a set of traces. A trace σ is *L-stable for* Σ if for every L-sequential ρ such that $\sigma\rho \in \Sigma$, there is no $a \in \sigma$ and $b \in \rho$ such that (a, b) is an L-race.

Transactional L-Sequentiality and L-Stability. Transaction b is *contiguous* if $\langle b{:}s\,\mathrm{B}\rangle \xrightarrow{\text{index}} \langle c{:}t\rangle$ and $s \neq t$ imply that either $\langle \mathrm{C}b\rangle \xrightarrow{\text{index}} c$, $\langle \mathrm{A}b\rangle \xrightarrow{\text{index}} c$, or there are no actions of s after c, i.e., $c \xrightarrow{\text{index}} \langle d{:}s'\rangle$ implies $s \neq s'$.

Note that contiguity allows multiple live transactions.

A trace is *transactionally L-sequential* if every action is L-sequential and every transaction is contiguous.

A trace σ is *transactionally L-stable for* Σ if it is L-stable for Σ, every transaction is both contiguous and resolved, and there is no $\beta \in \sigma$, $\sigma\rho \in \Sigma$, and $\alpha \in \rho$ such that α touches a variable in L and $\alpha \cdots^{\text{xrw}} \beta$.

The last condition ensures that a stable state is "future proof" by making all new conflicting transactions serialize afterwards.

Closure Conditions on Programs. The SC-LTRF theorem requires that we relate an arbitrary execution to one that is transactionally L-sequential. To ensure that such an execution exists, we assume that the semantics of programs is closed under certain operations.

We first give some preliminary definitions.

Let $\overset{\text{act}}{\sim}$ relate actions with the same thread and location:

$$\langle a{:}s\,\mathrm{W}x\,v\,q\rangle \overset{\text{act}}{\sim} \langle a'{:}s'\,\mathrm{W}x'\,v'\,q'\rangle \text{ if } a = a',\ s = s' \text{ and } x = x'$$
$$\langle a{:}s\,\mathrm{R}x\,v\,q\rangle \overset{\text{act}}{\sim} \langle a'{:}s'\,\mathrm{R}x'\,v'\,q'\rangle \text{ if } a = a',\ s = s' \text{ and } x = x'$$

A set Σ of traces is *sequentially-closed* if whenever a trace $\sigma\alpha \in \Sigma$ includes a Loc-weak action α, there exists a Loc-sequential action $\alpha' \overset{\text{act}}{\sim} \alpha$ such that $\sigma\alpha' \in \Sigma$.

For $a \in \sigma$, let $\sigma \downarrow a$ be the subsequence of σ obtained by removing all the events that causally follow a:

$$b \notin (\sigma \downarrow a) \text{ iff } a\ (\xrightarrow{\text{hb}} \cup \xrightarrow{\text{lwr}} \cup \cdots^{\text{xrw}})^+\ b$$

We say that a set of traces Σ is *causally closed* iff for any $\sigma \in \Sigma$, for any $a \in \sigma$, $\sigma \downarrow a \in \Sigma$.

Intuitively, $\sigma \downarrow a$ removes "causal upclosure" of a from σ. Significantly, if (b, α) is an L-race in $\sigma\alpha$, then $b \in \sigma\alpha \downarrow \alpha$. This property does not hold for the "causal downclosure."

For any consistent trace σ, we say that ρ is an *order-preserving permutation of* σ if ρ is a well-formed permutation of σ and $\xrightarrow{\text{po}}_\rho = \xrightarrow{\text{po}}_\sigma$.

If a trace is consistent, then any order-preserving permutation is also consistent, since the derived orders coincide. In addition, any consistent trace has an order-preserving permutation with contiguous transactions. We say that Σ is valid as the *semantics of a program* if (1) every $\sigma \in \Sigma$ is consistent, (2) Σ is sequentially closed, (3) Σ is causally closed, and (4) Σ is closed under order preserving permutation.

SC-LTRF. With these definitions, our theorem is as follows. The theorem establishes that any L-race can be discovered by a *sequential trace* with *contiguous transactions*.

Theorem 4.1 (SC-LTRF). *Fix Σ to be the semantics of a program. Fix $\sigma\rho\alpha \in \Sigma$ such that*

- *σ is transactionally L-stable,*
- *ρ is transactionally L-sequential in $\sigma\rho$,*
- *ρ has no L-races in $\sigma\rho$, and*
- *α is L-weak in $\sigma\rho\alpha$.*

Then, there are $b \in \rho$, $\alpha' \overset{\text{act}}{\sim} \alpha$ and $\sigma\rho'\alpha' \in \Sigma$ such that:

- *$\rho'\alpha'$ is transactionally L-sequential in $\sigma\rho'\alpha'$, and*
- *(b, α') is an L-race in $\sigma\rho'\alpha'$.*

With respect to the SC-LDRF theorem in [9], the SC-LTRF result differs in that we allow $\rho' \neq \rho$ and use the *transactional* variants of L-stability and L-sequentiality, which require that we only consider traces with contiguous transactions. In an L-stable trace, all transactions must also be resolved. In the degenerate case, with only contiguous committed singleton transactions, the definitions of SC-LDRF and SC-LTRF coincide.

For example, consider the (IRIW) program from the introduction. Reasoning sequentially, we know that we cannot read 1 followed by 0 in both threads. SC-LDRF validates this reasoning for concurrent executions. Likewise, the publication and privatization examples from the introduction have the expected behavior. As a further example in this vein, consider the following program.

$$\begin{array}{l} \text{atomic}_a\,\{\,\text{if } !y \text{ then while } x \text{ do skip}\,\} \\ \parallel \text{atomic}_b\,\{\,y := 1\,\};\; x := 1 \end{array}$$

If it is possible for a to read 0 for y and then 1 for x, then a becomes a *doomed transaction*, which can never commit. By sequential reasoning, this is impossible, and therefore, by SC-LTRF, it is impossible in our model.

It is worth emphasizing that the SC-LTRF theorem includes aborted and live transactions, and thus guarantees opacity. In addition, the following result shows that aborted transactions can be ignored.

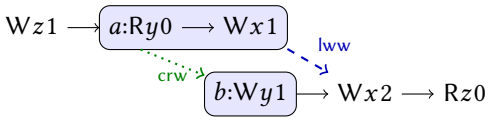
Theorem 4.2. *If σ is consistent then so is σ with aborted transactions removed.*

5 Implementation Model

An optimization is *valid* as long as it creates no new behaviors. As noted in §2, LDRF disables reads from being reordered with later writes. Thus we cannot transform $r := z$; $x := 1$ to $x := 1$; $r := z$. Unfortunately, the reverse transformation *also* fails in our programmer model, due to the order created by HB_{ww}. Consider the following variant of privatization:

$$\begin{array}{l} z := 1;\; \text{atomic}_a\,\{\,\text{if } !y \text{ then } x := 1\,\} \\ \parallel \text{atomic}_b\,\{\,y := 1\,\};\; x := 2;\; r := z \end{array}$$

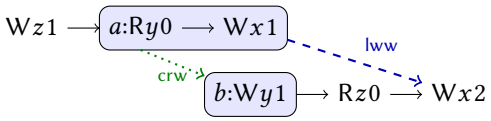
The second thread must read $\langle \text{R}z1 \rangle$. If not, we would obtain the following execution, which is disallowed by Observation.



(‡)

Note that $\langle \text{W}x2 \rangle \overset{\text{ww}}{\dashrightarrow} \langle \text{W}x1 \rangle$ is ruled out by Anti_{ww}, and so we must have $\langle \text{W}x1 \rangle \overset{\text{ww}}{\dashrightarrow} \langle \text{W}x2 \rangle$, as shown. By HB_{ww}, we have $\langle \text{W}x1 \rangle \xrightarrow{\text{hb}} \langle \text{W}x2 \rangle$, and thus by transitivity, $\langle \text{W}z1 \rangle \xrightarrow{\text{hb}} \langle \text{R}z0 \rangle$. Observation rules out the execution, since $\langle \text{R}z0 \rangle \overset{\text{rw}}{\dashrightarrow} \langle \text{W}z1 \rangle$.

However if we replace "$x := 2$; $r := z$" by "$r := z$; $x := 2$" in the program above, then the second thread may read $\langle \text{R}z0 \rangle$, since we no longer have $\langle \text{W}z1 \rangle \xrightarrow{\text{hb}} \langle \text{R}z0 \rangle$. The resulting allowed execution shows that the optimization is not valid:



In this section, we consider an "implementation" model that removes HB_{ww}. Since HB_{ww} is designed to allow non-racy privatization, it should not be surprising that privatization is racy in the implementation model. To enable the removal of such races, we add the new action $\langle s\text{Q}x \rangle$ to model a *quiescence fence* [36] for thread s on location x.

Note that our implementation model is still fairly abstract. We assume that the underlying transactional machinery provides order between transactions that have a direct dependency, as in the publication idiom. Quiescence fences are necessary only to provide order when there is no direct dependency, as in the privatization idiom.

$$\alpha \;::=\; \cdots \mid \langle a{:}s\text{Q}x \rangle \qquad \text{(Quiesence fence)}$$

A quiescence fence $\langle \text{Q}x \rangle$ may not be interleaved with a transaction that touches x. We therefore add the following requirement to *well-formedness*:

WF_{12}. If $\langle b{:}\text{B} \rangle \xrightarrow{\text{index}} \langle \text{Q}x \rangle$ then either $\langle \text{C}b \rangle \xrightarrow{\text{index}} \langle \text{Q}x \rangle$, $\langle \text{A}b \rangle \xrightarrow{\text{index}} \langle \text{Q}x \rangle$ or b neither reads not writes x.

In addition, quiescence fences create order. In the definition of *happens-before*, we replace HB_{ww} by the following.

$$\langle a{:}\text{C}b \rangle \xrightarrow{\text{hb}} \langle c{:}\text{Q}x \rangle \text{ if } a \xrightarrow{\text{index}} c \text{ and } b \text{ touches } x \qquad (\text{HB}_{\text{CQ}})$$
$$\langle c{:}\text{Q}x \rangle \xrightarrow{\text{hb}} \langle b{:}\text{B} \rangle \text{ if } c \xrightarrow{\text{index}} b \text{ and } b \text{ touches } x \qquad (\text{HB}_{\text{QB}})$$

Because we have removed HB_{ww}, we also drop Anti_{ww} from the definition of a *consistent* execution. The remaining definitions are unchanged in the implementation model.

Relating implementation and programmer models. The implementation model allows executions that are not allowed by the programmer model. Since Anti_{ww} is removed, Example 2.2 is allowed in the implementation model; however, there is no matching execution in the programmer model: If a precedes b, then the read of a is invalidated by

Observation. If b precedes a, the write-to-write order is invalidated by Coherence. Since $\mathrm{HB}_{\mathrm{ww}}$ is removed, (‡) is allowed in the implementation model; however, there is no matching execution in the programmer model: If a precedes b, then the read of z is invalidated by Observation. If b precedes a, then the read of y is invalidated by WF_{10}.

We say that σ has a *mixed race* if there is some $L \subseteq Loc$ such that σ includes an action in an L-race between a transactional write and a plain write.

The following lemma establishes that the implementation and programmer models coincide for programs without mixed races. Therefore, for *mixed-race free* programs in the implementation model, SC-LTRF holds. Khyzha et al. [22] establish a similar result for global TRF.

Lemma 5.1. *Let σ be an execution in the implementation model without mixed races. Let ρ be the induced execution in the programmer model obtained by dropping all the quiescence fences in σ. If σ is consistent, then so is ρ.*

Suborders. The quiescent fence $\langle \mathrm{Q}x \rangle$ has the same ordering properties as a committed transaction that writes x: $\langle a{:}\mathrm{B} \rangle \langle \mathrm{Q}x \rangle \langle \mathrm{C}a \rangle$. For the purpose of studying compiler optimizations, we encode quiescent fences thusly as writing transactions. With this convention, we do not mention $\langle \mathrm{Q}x \rangle$ explicitly in the following development. The treatment follows [9] closely, including much of the notation and proofs. We need only adapt their definitions to work up to $\overset{\mathsf{tx}}{\simeq}$.

Let $TAct = \{\langle \mathrm{B} \rangle, \langle \mathrm{C} \rangle, \langle \mathrm{A} \rangle\}$. Define the following subsets of $\xrightarrow{\mathsf{po}} \setminus (Act \times TAct \cup TAct \times Act)$, i.e., the portion of $\xrightarrow{\mathsf{po}}$ that does not involve the transactional boundaries. In the following definitions, we quantify universally over $a, b \in Act \setminus TAct$; all other actions are quantified existentially.

We say action *a conflicts with b* iff they access the same location at least one of a or b is a write.

$$a \xrightarrow{\mathsf{po\text{-}T}} b \text{ iff } a \xrightarrow{\mathsf{po}} b, a \not\overset{\mathsf{tx}}{\simeq} b, b \overset{\mathsf{tx}}{\simeq} \langle \mathrm{B} \rangle, \text{ and } b \overset{\mathsf{tx}}{\simeq} \langle \mathrm{W} \rangle$$
$$a \xrightarrow{\mathsf{poT\text{-}}} b \text{ iff } a \xrightarrow{\mathsf{po}} b, a \not\overset{\mathsf{tx}}{\simeq} b, \text{ and } a \overset{\mathsf{tx}}{\simeq} \langle \mathrm{B} \rangle$$
$$a \xrightarrow{\mathsf{poTT}} b \text{ iff } a \xrightarrow{\mathsf{poT\text{-}}} b \text{ and } a \xrightarrow{\mathsf{po\text{-}T}} b$$
$$a \xrightarrow{\mathsf{poRW}} b \text{ iff } a \xrightarrow{\mathsf{po}} b, a = \langle \mathrm{R} \rangle, \text{ and } b = \langle \mathrm{W} \rangle$$
$$a \xrightarrow{\mathsf{poCon}} b \text{ iff } a \xrightarrow{\mathsf{po}} b \text{ and } a \text{ conflicts with } b$$

The relations $\xrightarrow{\mathsf{po\text{-}T}}$, $\xrightarrow{\mathsf{poT\text{-}}}$, $\xrightarrow{\mathsf{poTT}}$ do not relate actions from the same transaction. $\xrightarrow{\mathsf{po\text{-}T}}$ is that subset of $\xrightarrow{\mathsf{po}}$ that ends in a transactional action of a writing transaction; $\xrightarrow{\mathsf{poT\text{-}}}$ is the subset of $\xrightarrow{\mathsf{po}}$ that begins in a resolved transactional action; whereas $\xrightarrow{\mathsf{poTT}}$ is the subset of $\xrightarrow{\mathsf{po}}$ that begins and ends in transactional actions with target being a writing transaction. The targets of relations $\xrightarrow{\mathsf{poTT}}$ and $\xrightarrow{\mathsf{po\text{-}T}}$ are restricted to transactions that contain a write action; this restriction mirrors the treatment of read actions of volatiles in [9] and ensures that read-only transactions have greater flexibility in commuting earlier in program order. $\xrightarrow{\mathsf{poRW}}$ is that subset of $\xrightarrow{\mathsf{po}}$ between reads and writes, not necessarily of the same location. $\xrightarrow{\mathsf{poCon}}$ restricts $\xrightarrow{\mathsf{po}}$ to conflicting actions.

In the supplementary material for this paper, we describe an equivalent definition of consistency that uses only these suborders instead of the full $\xrightarrow{\mathsf{po}}$. This characterization of consistency is useful for proving the correctness of the optimizations enumerated in the next subsection.

Compiler optimizations. Consider a program transformation $P \rhd Q$, where Q is a program obtained from P by reordering its statements. To validate the transformation, for any execution ρ of Q, we must associate a corresponding execution σ of P. We consider three flavors.

In the first method, the transformation is correct if there is no change in transactional actions, and

$$\begin{aligned}&(\xrightarrow{\mathsf{po\text{-}T}}_{\sigma}, \xrightarrow{\mathsf{poT\text{-}}}_{\sigma}, \xrightarrow{\mathsf{poTT}}_{\sigma}, \xrightarrow{\mathsf{poRW}}_{\sigma}, \xrightarrow{\mathsf{poCon}}_{\sigma})\\ = &(\xrightarrow{\mathsf{po\text{-}T}}_{\rho}, \xrightarrow{\mathsf{poT\text{-}}}_{\rho}, \xrightarrow{\mathsf{poTT}}_{\rho}, \xrightarrow{\mathsf{poRW}}_{\rho}, \xrightarrow{\mathsf{poCon}}_{\rho})\end{aligned}$$

This allows, for example, the reordering of independent writes and of independent reads. Dolan et al. [9] show how to prove the validity of some peephole optimizations using this flexibility: redundant load, store forwarding, dead store elimination, common subexpression elimination, constant propagation and loop invariant code motion. We show that:

$$P;\ \mathsf{atomic}\ \{\ Q\ \} \rhd \mathsf{atomic}\ \{\ Q\ \};\ P$$

if Q is read-only, P is write-only and there are no conflicts between P, Q. For correctness, note that $\xrightarrow{\mathsf{poTT}}$ and $\xrightarrow{\mathsf{po\text{-}T}}$ relations do not target read-only transactions. The absence of conflict between P, Q ensures the preservation of $\xrightarrow{\mathsf{poCon}}$. Moreover, $\xrightarrow{\mathsf{poRW}}$ is preserved because P is write only.

Secondly, we validate transformations, such as the roach motel optimization, where the only change is increase in the scope of transactions; i.e, when P and Q are nontransactional:

$$P;\ \mathsf{atomic}\ \{\ R\ \};\ Q \rhd \mathsf{atomic}\ \{\ P;\ R;\ Q\ \}.$$

Given ρ from atomic $\{\ P;\ R;\ Q\ \}$, we establish the consistency of the corresponding σ from P; atomic $\{\ R\ \}$; Q by showing that all relevant orders of σ are contained in those of ρ.

Thirdly, we validate the fusion of adjacent transactions:

$$\mathsf{atomic}\ \{\ P\ \};\ \mathsf{atomic}\ \{\ Q\ \} \rhd \mathsf{atomic}\ \{\ P;\ Q\ \}.$$

Given ρ from atomic $\{\ P;\ Q\ \}$, we build σ for atomic $\{\ P\ \}$; atomic $\{\ Q\ \}$ by adding two adjacent transactional events. On the other hand, the converse transformation is not validated. This is because we need to remove the two extra events to build a witness execution of atomic $\{\ P;\ Q\ \}$ from a given execution of atomic $\{\ P\ \}$; atomic $\{\ Q\ \}$. These events are not necessarily adjacent; so, the validity of the constructed execution cannot be established in general.

We can similarly establish that empty transactions can be elided, i.e.,

$$P;\ \mathsf{atomic}\{\};\ Q \rhd P;\ Q.$$

6 Compilation

Dolan et al. [9] show that the LDRF memory model can be compiled efficiently to both x86-TSO and AArch64/ARMv8.

Compilation of LDRF to x86-TSO requires no additional fencing. Therefore non-volatile reads/writes execute with native performance.

Because ARMv8 allows load buffering (which is disallowed by LDRF), compilation to ARMv8 requires some fencing, even for non-volatile reads/writes. [9] discusses two compilation schemes and studies their performance on several benchmarks with differing patterns of access. The performance penalty is 2.5% for one compilation strategy and 0.6% for the other. These results demonstrate that non-volatile access is not appreciably slowed by the insertion of fences to prevent load buffering.

The compilation results for plain variables carry over to our model, which differs from [9] primarily in the style of synchronization: [9] uses volatile variables, whereas we use transactions. In both x86-TSO and ARMv8 models, there are fences before and after successful transactions (see [6]), making the fencing behavior similar to that of volatile variables.

Both x86-TSO and ARMv8 validate our *implementation* model, assuming we include fences to prevent load-buffering in ARMv8, as described above.

In x86-TSO, $\overset{\text{crw}}{\cdots\!\rightarrow}$ order is included in $\overset{\text{hb}}{\longrightarrow}$. Thus, it is straightforward to establish that x86-TSO validates even the strongest variant of our *programmer* model, which includes HB_{WW}, HB_{RW}, HB_{WR} and their prime variants. Like our programmer model, x86-TSO validates privatization (Example 2.1). Like models that include ANTI'_{RW}, x86-TSO imposes publication by antidependence (Example 3.1). Neither of these examples require quiescent fences on x86-TSO.

It is not immediately obvious whether ARMv8 realizes our *programmer* model. In ARMv8, $\overset{\text{ob}}{\longrightarrow}$ plays the role of $\overset{\text{hb}}{\longrightarrow}$. The $\overset{\text{crw}}{\cdots\!\rightarrow}$ relation is included in $\overset{\text{ob}}{\longrightarrow}$ when the source and target come from different threads, known as *external from-read*. As a result, ARMv8 gives the same strong result as x86-TSO for Examples 2.1 and 3.1.

We expect that software transactional memories will realize the *implementation* model of §5, rather than the *programmer* model. As a result, it will be necessary for either the programmer or compiler to insert quiescent fences in order to realize our *programmer* model. Our results provide a correctness criterion: when are there sufficient fences to guarantee the absence of data races in the *implementation* model. As we discuss in §7, our work on placing quiescent fences is compatible with, and builds on, the extensive literature exploring this topic.

7 Related Work and Conclusions

Transactions [12, 18, 33] are motivated by the issues that arise with lock-based programming. See [14, 16, 17, 23] for textbook-style presentations. Hardware transactional models that integrate with relaxed memory are available for Pentium, Power and ARMV8 (in design) [5, 6, 10]. Software transactional memory achieves transactional guarantees limitations of the "bounded" and "best-effort" hardware transactional model, e.g., the C++ design of transactions [29] in C11 [4], Haskell transactions in GHC 6.4, experimental designs for Java [20] and C# [2].

Inspired by Dalessandro et al. [7] and Grossman et al. [13], we use memory orders to integrate transactions into the relaxed memory model of Dolan et al. [9].

In order to permit compiler optimizations, the LDRF model of [9] is more liberal than sequential consistency. Yet it eschews the speculative reads found in many models [19, 21, 25]. There is a rich design space for such "intermediate" models. Ou and Demsky [30] includes a survey of this work.

Transactional sequential consistency is similar to the the strong semantics [1], StrongBasic semantics [28], strong isolation [17], and transactional memory with store atomicity by [24]. Opacity [15, 16] and *TMS2* [8] treat aborted transactions in this context (see [11] for a survey).

Our model of SC-TDRF replaces the global real-time order by memory orders. We exploit the LDRF framework [8] to achieve a modular form of LTRF that is insensitive to races that are spatially and temporally isolated from the transactions under consideration. LDRF is defined operationally in [9], using machine states. We give an axiomatic account. The two approaches are equivalent if every machine state is derivable from the initial state.

Our results in §5 show that our model does not suffer from "optimization obstruction" [35]. Prior work, e.g., [22, 34, 35], requires that programmers place quiescence fences in order to guarantee safe privatization. Our low level model illustrates the correctness criteria for such techniques.

In Spear et al. [35], transactions can optionally be marked with annotations corresponding to publishing/privatizing transactions. The weakest ordering $\overset{\text{sfs}}{\longrightarrow}$ in [35] is the smallest transitive relation that includes transactional ordering and ensures that $a \overset{\text{sfs}}{\longrightarrow} c$ in the cases when: (1) a is an acquire transaction, $a \overset{\text{po}}{\longrightarrow} c$, and $a \overset{\text{tx}}{\not\rightarrow} c$, or (2) there is some release transaction b such that $a \overset{\text{po}}{\longrightarrow} b$ and either $b \overset{\text{lwr}}{\longrightarrow} c$ or a is transactionally ordered before c. There are two kinds of fences in the implementation level model of §5, namely the explicit quiescence fences $\langle \text{Q}x \rangle$, and the implicit memory fences arising from our transactional abstraction. In each case, we can deduce $a \overset{\text{sfs}}{\longrightarrow} c$, thus showing that our requirements for synchronization are no stronger than those of [35].

Our treatment of the implementation model is inspired by Khyzha et al. [22]. They divide actions into request/response pairs such that transactional response actions may abort. Our treatment is more abstract. We record all failed requests using a single abort action. Our commit action corresponds to the commit request in [22]. All of our other actions correspond to a response in [22].

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. 2011. Semantics of Transactional Memory and Automatic Mutual Exclusion. *ACM Trans. Program. Lang. Syst.* 33, 1, Article 2 (Jan. 2011), 50 pages.

[2] M. Abadi, T. Harris, and M. Mehrara. 2009. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP*, D. A. Reed and V. Sarkar (Eds.). ACM, 185–196.

[3] S. V. Adve and M. D. Hill. 1990. Weak Ordering—a New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*. ACM, New York, NY, USA, 2–14.

[4] H.-J. Boehm and S. V. Adve. 2008. Foundations of the C++ concurrency memory model. In *PLDI*. ACM, 68–78.

[5] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Q. Le. 2013. Robust architectural support for transactional memory in the Power architecture. In *ISCA*, A. Mendelson (Ed.). ACM, 225–236.

[6] N. Chong, T. Sorensen, and J. Wickerson. 2018. The semantics of transactions and weak memory in x86, Power, ARM, and C++. In *PLDI*. ACM, 211–225.

[7] L. Dalessandro, M. L. Scott, and M. F. Spear. 2010. Transactions As the Foundation of a Memory Consistency Model. In *DISC (LNCS)*. Springer-Verlag, Berlin, Heidelberg, 20–34.

[8] S. Doherty, L. Groves, V. Luchangco, and M. Moir. 2013. Towards formally specifying and verifying transactional memory. *Formal Asp. Comput.* 25, 5 (2013), 769–799.

[9] S. Dolan, K. C. Sivaramakrishnan, and A. Madhavapeddy. 2018. Bounding data races in space and time. In *PLDI*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 242–255.

[10] B. Dongol, R. Jagadeesan, and J. Riely. 2018. Transactions in relaxed memory architectures. *PACMPL* 2, POPL (2018), 18:1–18:29.

[11] D. Dziuma, P. Fatourou, and E. Kanellou. 2015. *Consistency for Transactional Memory Computing*. Springer International Publishing, Cham, 3–31.

[12] J. E. Gottschlich and H.-J. Boehm. 2013. Generic programming needs transactional memory. In *The 8th ACM SIGPLAN Workshop on Transactional Computing*.

[13] D. Grossman, J. Manson, and W. Pugh. 2006. What do high-level memory models mean for transactions?. In *Memory System Performance and Correctness*. ACM, New York, NY, USA, 62–69.

[14] D. Grossman, V. Menon, S. Srinivas, and C. Zilles. 2007. Transactional Memory in Managed Runtimes - Hardware/Software View. https://www.microarch.org/micro40

[15] R. Guerraoui and M. Kapalka. 2008. On the Correctness of Transactional Memory. In *PPoPP*. ACM, New York, NY, USA, 175–184.

[16] R. Guerraoui and M. Kapalka. 2010. *Principles of Transactional Memory*. Morgan & Claypool Publishers.

[17] T. Harris, J. Larus, and R. Rajwar. 2010. *Transactional Memory, 2nd edition*. Morgan & Claypool Publishers.

[18] M. Herlihy and J. E. B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, A. J. Smith (Ed.). ACM, 289–300.

[19] R. Jagadeesan, C. Pitcher, and J. Riely. 2010. Generative Operational Semantics for Relaxed Memory Models. In *ESOP*. 307–326.

[20] S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. 2005. A transactional object calculus. *Science of Computer Programming* 57, 2 (2005), 164 – 186.

[21] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 175–189.

[22] A. Khyzha, H. Attiya, A. Gotsman, and N. Rinetzky. 2018. Safe privatization in transactional memory. In *PPOPP*. ACM, 233–245.

[23] J. Larus and C. Kozyrakis. 2008. Transactional Memory. *Commun. ACM* 51, 7 (July 2008), 80–88.

[24] J.-W. Maessen and Arvind. 2007. Store Atomicity for Transactional Memory. *Electronic Notes in Theoretical Computer Science* 174, 9 (2007), 117 – 137. Proceedings of the Thread Verification Workshop (TV 2006).

[25] J. Manson, W. Pugh, and S. V. Adve. 2005. The Java memory model. In *POPL*. 378–391.

[26] M. Martin, C. Blundell, and E. Lewis. 2006. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Comput. Archit. Lett.* 5, 2 (July 2006), 17–17.

[27] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. 2008. Practical Weak-atomicity Semantics for Java Stm. In *SPAA*. ACM, New York, NY, USA, 314–325.

[28] K. F. Moore and D. Grossman. 2008. High-level small-step operational semantics for transactions. In *POPL*, G. C. Necula and P. Wadler (Eds.). ACM, 51–62.

[29] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. 2008. Design and Implementation of Transactional Constructs for C/C++. *SIGPLAN Not.* 43, 10 (Oct. 2008), 195–212.

[30] Peizhao Ou and Brian Demsky. 2018. Towards Understanding the Costs of Avoiding Out-of-thin-air Results. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 136 (Oct. 2018), 29 pages.

[31] W. Pugh. 1999. Fixing the Java Memory Model. In *Proceedings of the ACM 1999 Conference on Java Grande (JAVA '99)*. ACM, New York, NY, USA, 89–98.

[32] A. Raad, O. Lahav, and V. Vafeiadis. 2018. On Parallel Snapshot Isolation and Release/Acquire Consistency. In *ESOP (Lecture Notes in Computer Science)*, Vol. 10801. Springer, 940–967.

[33] N. Shavit and D. Touitou. 1995. Software Transactional Memory. In *PODC*. ACM, New York, NY, USA, 204–213.

[34] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. 2007. Enforcing isolation and ordering in STM. In *PLDI*, J. Ferrante and K. S. McKinley (Eds.). ACM, 78–88.

[35] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. 2008. Ordering-Based Semantics for Software Transactional Memory. In *PODS*, T. P. Baker, A. Bui, and S. Tixeuil (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 275–294.

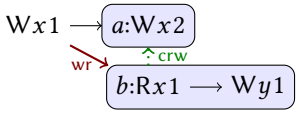
[36] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. 2007. Privatization Techniques for Software Transactional Memory. In *PODC*. ACM, New York, NY, USA, 338–339.

A Proof of SC-LTRF Theorem

We begin with an example to explain the last condition in the definition of transactional L-stability.

Example A.1. Recall the definition of transactionally L-stability: A trace is *transactionally L-stable for Σ* if it is L-stable for Σ, every transaction is both contiguous and resolved, and there are no $\sigma\rho \in \Sigma$, $\beta \in \sigma$, and $\alpha \in \rho$ such that α touches a variable in L and $\alpha \xrightarrow{\text{xrw}} \beta$.

To see the need for the last requirement, consider the following consistent execution:



Take $L = \{y\}$ and consider the execution in σ contains the top thread, ρ contains the read of the bottom thread, and α is the write. Ignoring initialization, we have $\sigma = \langle s\mathrm{W}x1\rangle\ \langle a{:}s\mathrm{B}\rangle\ \langle s\mathrm{W}x2\rangle\ \langle s\mathrm{C}a\rangle$, $\rho = \langle b{:}t\mathrm{B}\rangle\ \langle t\mathrm{R}x1\rangle$, and $\alpha = \langle t\mathrm{W}y1\rangle$.

This particular decomposition invalidates the theorem, since we must remove a from σ in order to linearize b, yet a occurs in σ.

The last requirement forbids this decomposition. In considering the trace where a occurs before b, we must include a in ρ, not σ.

In order to prove the theorem, we first establish several lemmas. The first two concern causal closure. Recall that $\sigma \downarrow a$ is the subtrace of σ that discards all causal dependents of a, defined as:

$$b \notin (\sigma \downarrow a) \text{ iff } a\ (\xrightarrow{\text{hb}} \cup \xrightarrow{\text{lwr}} \cup \cdots\!\!\xrightarrow{\text{xrw}})^+\ b$$

In the rest of the appendix, we use the notation $\sigma \downarrow \rho$ to stand for:

$$b \notin (\sigma \downarrow \rho) \text{ iff } (\forall a \in \rho)\ a\ (\xrightarrow{\text{hb}} \cup \xrightarrow{\text{lwr}} \cup \cdots\!\!\xrightarrow{\text{xrw}})^+\ b$$

It is immediate that $\sigma \downarrow \rho$ is invariant under permutations of ρ.

Note that $a \in \sigma \downarrow a$. In the case where a is transactional, the effect of $\sigma \downarrow a$ is to remove all the dependent transactions that read from the transaction, and also the anti-dependent transactions. Thus, for any transactional $b \overset{\text{tx}}{\sim} a$ that is a read, there are no transactional conflicting writes in $\sigma \downarrow a$ with a later timestamp.

The first lemma shows that σ is included in $\sigma\rho\alpha \downarrow \alpha$ whenever it is stable.

Lemma A.2. *Suppose σ is transactionally L-stable ρ is transactionally L-sequential in $\sigma\rho$, and α touches a location in L. Then σ is a prefix of $\sigma\rho\alpha \downarrow \alpha$.*

Proof. We show that for all $b \in \sigma$

$$\neg(\alpha\ (\xrightarrow{\text{hb}} \cup \xrightarrow{\text{lwr}} \cup \cdots\!\!\xrightarrow{\text{xrw}})^+\ b)$$

It suffices to prove $\forall c \in \sigma\rho\alpha$, if $a \in \sigma$ is such that

$$c\ (\xrightarrow{\text{hb}} \cup \xrightarrow{\text{lwr}} \cup \cdots\!\!\xrightarrow{\text{xrw}})\ a,$$

then $c \in \sigma$. We proceed by cases:

- $c \cdots\!\!\xrightarrow{\text{xrw}} a$. By the assumption that ρ is transactionally L-sequential. (This requires the assumption that α touches a location in L.)
- $c \xrightarrow{\text{lwr}} a$. Since σ is a prefix of $\sigma\rho\alpha$, the result follows by WF_8.
- $c \xrightarrow{\text{hb}} a$. There are two sub cases.
  - $c \xrightarrow{\text{hb}} a$ by item $\mathrm{HB}_{\mathrm{BASE}}$. The required result follows by WF_9–WF_{11}.
  - $c \xrightarrow{\text{hb}} a$ by item $\mathrm{HB}_{\mathrm{WW}}$. The required result follows by transactional L-stability of σ. □

The next lemma establishes that causal closure preserves transactional L-sequentiality.

Lemma A.3. *Suppose σ is transactionally L-stable ρ is transactionally L-sequential in $\sigma\rho$, and α touches a location in L. Then $\sigma\rho\alpha \downarrow \alpha = \sigma\rho'\alpha$, where $\rho'\alpha$ is transactionally L-sequential in $\sigma\rho$.*

Proof. If $\alpha \notin \sigma$, then the result is trivial, since $\sigma\downarrow\alpha = \sigma$. Thus, assume $\alpha \in \sigma$. Using the Lemma A.2, we know that $\sigma\rho\alpha \downarrow \alpha$ includes σ. Thus we can fix ρ' so that $\sigma\rho\alpha \downarrow \alpha = \sigma\rho'\alpha$.

First, we show that $\sigma\rho'\alpha$ is well-formed. WF_1–WF_5, and WF_7–WF_8 follow from the well-formedness of σ. WF_6 follows since $\pi \downarrow \alpha$ is closed under the predecessors of $\xrightarrow{\text{lwr}}$, for any π. WF_9 and WF_{10} follow since, $\pi \setminus (\pi \downarrow \alpha)$ is closed under $\overset{\text{tx}}{\sim}$. WF_{11} follows, since it is preserved under removal of actions.

Consistency of $\sigma\rho'\alpha$ follows from the consistency of $\sigma\rho\alpha$ since all relations on $\sigma\rho'\alpha$ are subrelations of $\sigma\rho\alpha$.

Transactional L-sequentiality of $\sigma\rho'\alpha$ follows from transactional L-sequentiality of $\sigma\rho\alpha$. □

The next lemma establishes that that any L-weak action participates in an L-race. The proof mirrors the last two paragraphs of the proof theorem 13 of [9]. Instead of reasoning operationally, we use the consistency axioms Coherence and Causality.

Lemma A.4. *Suppose σ is L-stable, ρ is L-sequential and $\sigma\rho\langle c\rangle \in \Sigma$. If c is L-weak then there exists some $b \in \rho$ such that (b, c) is an L-race.*

Proof. Suppose c is L-weak. Then, there exists a write action $b \xrightarrow{\text{index}} c$ such that either

- $c \dashrightarrow b$, or
- $a \xrightarrow{\text{wr}} c$ and $a \dashrightarrow b$; thus, $c \cdots\!\!\xrightarrow{\text{rw}} b$.

If $b \xrightarrow{\text{hb}} c$, we have a contradiction, either because

- $c \dashrightarrow b$ contradicts the irreflexivity of $(\xrightarrow{\text{hb}}; \dashrightarrow)$, or
- $c \cdots\!\!\xrightarrow{\text{rw}} b$ contradicts the irreflexivity of $(\xrightarrow{\text{hb}}; \cdots\!\!\xrightarrow{\text{lrw}})$.

So, (b, c) is an L-race. Further, b cannot be in σ since σ is L-stable; therefore, b must be in ρ, as required. □

The next lemma says that every execution has an order-preserving permutation with contiguous transactions. The proof formalizes the following argument: All the ordering

between transactional actions is reflected in the causality order. Furthermore, two actions in the same transaction are treated identically by the causality order. Consequently, since the causality order is acyclic, we can use a linearization of it to achieve contiguity in transactions.

Lemma A.5. *Let Σ be the semantics of a program, and fix $\sigma\rho \in \Sigma$. Suppose that all transactions of σ are contiguous and no transactions of σ are live in σ. Then there exists an order-preserving permutation $\sigma\pi$ of $\sigma\rho$ such that $\sigma\pi \in \Sigma$ and $\sigma\pi$ has contiguous transactions.*

Proof. By Causality, $(\xrightarrow{\text{hb}} \cup \xrightarrow{\text{lwr}} \cup \cdots\!\!\rightarrow^{\text{xrw}})$ is acyclic. Thus, we can extend $(\xrightarrow{\text{hb}} \cup \xrightarrow{\text{lwr}} \cup \cdots\!\!\rightarrow^{\text{xrw}})^*$ to a total order over the actions of $\sigma\rho$. Fix such a total order (with the initializing begin transaction as minimal element), and let R be the sub-order that includes only nontransactional actions and begin actions. We extend R to a total order over the actions of $\sigma\rho$ as follows. Define $a \trianglelefteq b$ when one of the following holds:

$$a \in \sigma \wedge b \in \rho \tag{1}$$

$$a \overset{\text{tx}}{\sim} a' \; R \; b' \overset{\text{tx}}{\sim} b \tag{2}$$

$$a \overset{\text{tx}}{\sim} b \wedge a \xrightarrow{\text{index}}_{\sigma\rho} b \tag{3}$$

Condition (1) ensures that the actions in σ are ordered before those in ρ. Condition (2) ensures that the actions in a transaction of $\sigma\rho$ are treated identically by $\trianglelefteq$ with respect actions outside the transaction—recall that $\overset{\text{tx}}{\sim}$ relates each nontransactional action to itself. Condition (3) forces order within a transaction of $\sigma\rho$ to coincide with the order from $\xrightarrow{\text{index}}_{\sigma\rho}$.

It is clear that $\trianglelefteq$ induces a total order on the actions $\sigma\rho$ with contiguous transactions. Supposing that the trace ordered by $\trianglelefteq$ is well formed, then it trivial to show that it is consistent, since no orders are changed. Because the semantics of a program must be closed with respect to order-preserving permutation, we further have that the trace belongs to Σ.

Thus, to prove the lemma it suffices to show that the trace ordered by $\trianglelefteq$ is well-formed. We consider each of the well-formedness criteria given in §2.

WF_1 follows from the choice of R.

WF_2–WF_4 and WF_6–WF_7 follow from the well-formedness of $\sigma\rho$.

WF_5 holds due to well formedness of $\sigma\rho$ and (3).

If both actions are nontransactional, WF_8 follows from well-formedness of $\sigma\rho$. If both are transactional, it follows because $\xrightarrow{\text{cwr}}$ is included in $\xrightarrow{\text{hb}}$. Suppose the write is transactional and the read is not. Then the begin is ordered with respect to the read in the lifted relation $\xrightarrow{\text{lwr}}$. Using (2) and (3), the result holds. The argument is symmetric for the case where the read is transactional and the write is not.

For WF_9, if a, b are conflicting transactional writes, then $a \xrightarrow{\text{hb}} b$ or $b \xrightarrow{\text{hb}} a$. In the former case, $a \trianglelefteq b$, by definition of R. The case for $b \xrightarrow{\text{hb}} a$ is symmetric.

For WF_{10}, let a, b be conflicting transactional writes such that $a \dashrightarrow^{\text{ww}} b$ and let $a \xrightarrow{\text{wr}} c$. Thus, $c \cdots\!\!\rightarrow^{\text{rw}} b$. Since c is also transactional we have $c \cdots\!\!\rightarrow^{\text{xrw}} b$. Thus, $c \trianglelefteq b$ by the definition of R.

For WF_{11}, let b be transactional and $a \xrightarrow{\text{wr}} b$ and $a \dashrightarrow^{\text{ww}} c$ and $c \overset{\text{tx}}{\sim} b$. If $c \trianglelefteq b$, then $c \xrightarrow{\text{index}} b$ contradicting WF_{11} on $\sigma\rho$. □

The next lemma shows that races are preserved by delaying the timestamp of writes. The intuition is that delaying the timestamp of a write can only decrease happens before. Note that only the timestamp of the last write is increased, and since timestamps are rationals, it is straightforward to change a timestamp so that the execution under consideration remains consistent.

The key step in the proof is the inductive case for HB_{ww}, which requires Anti_{ww}.

Lemma A.6. *Let $\sigma = \pi\alpha$ be a consistent execution such that (β, α) is an L-race in σ between two writes. Let $\rho = \pi\alpha'$ where $\alpha' \overset{\text{act}}{\sim} \alpha$ and α' has a later timestamp.*

Then ρ is a consistent and (β, α') is an L-race in ρ.

Proof. Since $\rho = \pi\alpha'$ and $a \xrightarrow{\text{hb}}_{\rho} c$ implies $a \xrightarrow{\text{index}}_{\rho} c$, it is not possible that $\alpha' \xrightarrow{\text{hb}}_{\rho} c$ or $\alpha \xrightarrow{\text{hb}}_{\sigma} c$. We call this property *Terminal*.

We show that $a \xrightarrow{\text{hb}}_{\rho} c$ implies $a \xrightarrow{\text{hb}}_{\sigma} c$, for any a, c.

The proof proceeds by induction on the definition of $\xrightarrow{\text{hb}}$. The empty relation satisfies the hypothesis. For the inductive step, we have three cases. If $a \xrightarrow{\text{hb}}_{\rho} c$ and $\alpha' \neq c$, then $a \xrightarrow{\text{hb}}_{\pi} c$ and therefore $a \xrightarrow{\text{hb}}_{\sigma} c$. Thus, we need only consider cases where $\alpha = c$.

- For HB_{BASE}, note that $\xrightarrow{\text{init}}_{\sigma} = \xrightarrow{\text{init}}_{\rho}$ and $\xrightarrow{\text{po}}_{\sigma} = \xrightarrow{\text{po}}_{\rho}$. If α' is transactional, then, by construction, α must also be transactional. In this case, using *Terminal*, we deduce that $\xrightarrow{\text{cwr}}_{\sigma} = \xrightarrow{\text{cwr}}_{\rho}$. Since α and α' are transactional writes on the same variable, using *Terminal*, we deduce that $\dashrightarrow^{\text{cww}}_{\sigma} = \dashrightarrow^{\text{cww}}_{\rho}$. If α' is nontransactional, then modifying the timestamp of α has no effect on any of the relations in HB_{BASE}.
- HB_{TRANS} follows immediately by induction.
- For HB_{ww}, suppose that a and b are nonaborted and transactional, α' is plain, $a \dashrightarrow^{\text{lww}}_{\rho} \alpha'$, $a \cdots\!\!\rightarrow^{\text{crw}}_{\rho} b$ and $b \xrightarrow{\text{hb}}_{\rho} \alpha'$.
  We have $a \cdots\!\!\rightarrow^{\text{crw}}_{\pi} b$ and therefore $a \cdots\!\!\rightarrow^{\text{crw}}_{\sigma} b$.
  Since $\alpha \overset{\text{act}}{\sim} \alpha'$, we know that α and α' have the same name. Applying the induction hypothesis to $b \xrightarrow{\text{hb}}_{\rho} \alpha'$, we have $b \xrightarrow{\text{hb}}_{\sigma} \alpha$.
  Note that the timestamp of α cannot be less than that of a. If this were the case, then we would also have that the timestamp of α is less than that of a, and we would have $\alpha \dashrightarrow^{\text{lww}}_{\sigma} a$ and $a \cdots\!\!\rightarrow^{\text{crw}}_{\sigma} b \xrightarrow{\text{hb}}_{\sigma} \alpha$. Thus, σ would fail to be consistent by Anti_{ww}.
  Since the timestamp of α must be greater than that of a, we have $a \dashrightarrow^{\text{lww}}_{\sigma} \alpha$. Thus, by HB_{ww}, we have as $a \xrightarrow{\text{hb}}_{\sigma} \alpha$ required.

Well-formedness of ρ is immediate.

Consistency of ρ follows from $\xrightarrow{\text{hb}}_\rho \subseteq \xrightarrow{\text{hb}}_\sigma$, using the consistency of σ.

The raciness of (β, α') in ρ also follows from $\xrightarrow{\text{hb}}_\rho \subseteq \xrightarrow{\text{hb}}_\sigma$, using the from the fact that $\xrightarrow{\text{hb}}_\rho$ is included in $\xrightarrow{\text{hb}}_\sigma$, using the fact that (β, α) is an L-race in σ. □

The next lemma shows that races are preserved by delaying the timestamp of some reads. The intuition again is that delaying the timestamp can only decrease happens before. The sole case when delaying the timestamp of a read can actually increase happens before is when the read is transactional and the newly matched write is also transactional. The hypothesis of the following lemma rules out this problematic case.

Lemma A.7. *Let $\sigma = \pi\alpha$ be a consistent execution such that (β, α) is an L-race in σ, β is a write and α is a read. Let $\rho = \pi\alpha'$ where $\alpha' \overset{\text{act}}{\sim} \alpha$ and α' has a later timestamp.*

Suppose that the writes satisfying α and α' are nontransactional when α is transactional (and therefore α' is transactional).

Then ρ is a consistent and (β, α') is an L-race in ρ.

Proof. The proof is similar to the proof of Lemma A.6.

For HB_{BASE}, the result follows since the writes matching α, α' are not transactional when α and α' are transactional.

For rule HB_{WW}, the result follows since α and α' are not writes. □

Lemma A.8. *Fix Σ to be the semantics of a program. Fix $\sigma\rho\alpha \in \Sigma$ such that*

- *σ is transactionally L-stable,*
- *ρ is transactionally L-sequential in $\sigma\rho$, and*
- *ρ has no L-races in $\sigma\rho$.*

Then, there is $\sigma\rho'\alpha \in \Sigma$ such that

- *ρ' is transactionally L-sequential in $\sigma\rho'$,*
- *$\rho'\alpha$ has contiguous transactions, and*
- *ρ' is an order-preserving permutation of a subsequence of ρ.*

Proof. If α is non-transactional or a begin action, setting $\rho = \rho'$ meets the requirements. Thus we suppose that α is transactional, belonging to transaction a of thread s.

Let $\pi = \sigma\rho\alpha \downarrow a$. Let ρ' be derived from π by permuting the events of the open transaction a to the end.

This order preserving permutation establishes contiguity of transactions in $\rho'\alpha$.

Next, we show that $\sigma\rho'\alpha$ is well-formed. WF_1–WF_7 are immediate. WF_8 follows because the writes of a are only read by actions of a by WF_7. WF_9 and WF_{10} follow because ρ' is derived from π. WF_{11} is inherited from well-formedness of $\sigma\rho\alpha$.

Finally, we show that ρ' is L-sequential in $\sigma\rho'$. We proceed by contradiction. There are two cases to consider. Let c be an arbitrary action in ρ'.

- c touches a location in L and there is a $b \xrightarrow{\text{index}} c$ such that $c \dashrightarrow b$. Since $\sigma\rho\alpha$ is well formed, this can only happen if c is in open transaction a and c was before b in $\sigma\rho\alpha$.
  We reason by cases based on whether b is transactional.
  - If b is transactional, $c \xrightarrow{\text{hb}} b$; so $b \notin \pi$.
  - If b is not transactional. In this case, since $b \notin \pi$, we deduce that $\neg(c \xrightarrow{\text{hb}} b)$. So, since there are no data races in ρ, we deduce that $b \xrightarrow{\text{hb}} c$ which contradicts Coherence of $\sigma\rho$.
- c touches a location in L, $a \xrightarrow{\text{wr}} c$, and there is $b \xrightarrow{\text{index}} c$ such that $a \dashrightarrow b$. We reason by cases based on whether b is transactional.
  - If b is transactional, $c \xrightarrow{\text{xrw}} b$; so $b \notin \pi$.
  - If b is not transactional. In this case, since $b \notin \pi$, we deduce that $\neg(c \xrightarrow{\text{hb}} b)$. So, since there are no data races in ρ, we deduce that $b \xrightarrow{\text{hb}} c$ which contradicts Observation of $\sigma\rho$. □

We now turn to the theorem.

Theorem 4.1. *Fix Σ to be the semantics of a program. Fix $\sigma\rho\alpha \in \Sigma$ such that*

- *σ is transactionally L-stable,*
- *ρ is transactionally L-sequential in $\sigma\rho$,*
- *ρ has no L-races in $\sigma\rho$, and*
- *α is L-weak in $\sigma\rho\alpha$.*

Then, there are $b \in \rho$, $\alpha' \overset{\text{act}}{\sim} \alpha$ and $\sigma\rho'\alpha' \in \Sigma$ such that

- *$\rho'\alpha'$ is transactionally L-sequential in $\sigma\rho'\alpha'$, and*
- *(b, α') is an L-race in $\sigma\rho'\alpha'$.*

Proof. By Lemma A.8, we can assume without loss of generality that $\sigma\rho\alpha$ has contiguous transactions.

Choose b as follows. Since α is L-weak, by Lemma A.4, we know that there is some b such that (b, α) is an L-race. By the definition of stability, we know that b must occur in ρ.

Choose α' as follows. Since Σ is *sequentially-closed*, there must be a L-sequential action $\alpha' \overset{\text{act}}{\sim} \alpha$ such that $\sigma\rho\alpha' \in \Sigma$.

Choose ρ' as follows. By Lemma A.3, there is some ρ' such that $\sigma\rho\alpha \downarrow \alpha' = \sigma\rho'\alpha'$. Since Σ is *causally-closed*, we know that $\sigma\rho'\alpha' \in \Sigma$. Since ρ' is a subsequence of ρ, all transactions of ρ' are contiguous. By construction, using Lemma A.3, we know that $\rho'\alpha'$ is L-sequential in $\sigma\rho'\alpha'$. Thus, $\rho'\alpha'$ is transactionally L-sequential in $\sigma\rho'\alpha'$.

We need only show that (b, α') is an L-race. We proceed by cases.

- Suppose that α is a B, C, and A action. This is not possible since these actions are always L-sequential.
- Suppose that α is a write. The result follows from Lemma A.6.
- Suppose that α is a non transactional read. The result follows from Lemma A.7.
- Finally, suppose that α is a transactional read.
  The write matching α must be nontransactional. Otherwise WF_{10} guarantees that α would be L-sequential.

The write matching α' must be nontransactional. Otherwise it would follow α in $\xrightarrow{xrw}$, and thus must have been removed from the causal closure. (This case corresponds to executions illustrated at the beginning of the paragraph labelled "*From* D *to* T" on page 8.) Given that the fulfilling writes for α and α' are not transactional, the hypotheses of lemma A.7 are satisfied, yielding the required result. □

B Aborted Transactions

Theorem 4.2. *If σ is consistent then so is σ with aborted transactions removed.*

Proof. Let ρ be any well-formed and consistent trace. Then:

- ρ without a is well-formed in the case that $a = \langle \mathsf{R} \rangle$ or $a = \langle \mathsf{W} \rangle$ and a is not the source of an $\xrightarrow{xwr}$ edge.
- by WF_7, if $a = \langle \mathsf{W} \rangle$ is in an aborted transaction, any read of a is also in the same aborted transaction.
- ρ with $\langle \mathsf{B} \rangle$ (and any matching $\langle \mathsf{end} \rangle$) removed is also well-formed.

Let σ be a well-formed and consistent trace. Let us write $\sigma \setminus \mathsf{A}$ for σ with aborted transactions removed. By above observation, $\sigma \setminus \mathsf{A}$ is well-formed. Consistency of $\sigma \setminus \mathsf{A}$ follows from the consistency of σ because the relations on $\sigma \setminus \mathsf{A}$ are merely the restriction of those in σ to a subset of events. □

C Technical Development for §5

The intuition behind the proof of Lemma 5.1 is that the extra explicit ordering in an implementation race free execution compensates for the specified extra $\mathrm{HB}_{\mathrm{ww}}$ and $\textsc{Anti}_{\mathrm{ww}}$ axioms in the programmer model.

Lemma 5.1. *Let σ be an execution in the implementation model without mixed races. Let ρ be the induced execution in the programmer model obtained by dropping all the quiescence fences in σ. If σ is consistent, then so is ρ.*

Proof. Well-formedness of ρ is immediate.

Consistency of ρ follows if we can show that the orders in ρ agree with those in σ. Thus, it suffices to show that σ satisfies $\mathrm{HB}_{\mathrm{ww}}$ and $\textsc{Anti}_{\mathrm{ww}}$. We proceed as follows.

To show $\mathrm{HB}_{\mathrm{ww}}$, let c be plain, $a \xrightarrow{lww} c$, and $a \xrightarrow{crw} b \xrightarrow{hb} c$ in σ. Then, by implementation race freedom, we must have $a \xrightarrow{hb} c$, otherwise a and c would be racing.

To show $\textsc{Anti}_{\mathrm{ww}}$, suppose $a \xrightarrow{crw} b \xrightarrow{hb} c \xrightarrow{lww} a$ in σ. By implementation race freedom, we must have $c \xrightarrow{hb} a$. However, this leads to a cycle in $\xrightarrow{crw} \cup \xrightarrow{hb}$, contradicting the observation axiom of σ. □

Suborders We follow [9] in providing an alternate characterization of $\xrightarrow{hb}$ in the implementation model. Recall that the $\xrightarrow{hb}$ relation in the implementation model does not include $\mathrm{HB}_{\mathrm{ww}}$.

Let $\xrightarrow{swe} = (\xrightarrow{cwr} \cup \xrightarrow{cww}) \setminus \xrightarrow{po}$ be the external transactional communication relation, which captures the basic ingredients in the $\xrightarrow{hb}$ relation across threads, namely external transactional reads-from and external transactional coherence.

Let $\xrightarrow{hbe} = \xrightarrow{po\text{-}T}; (\xrightarrow{swe}; \xrightarrow{poTT})^\star; \xrightarrow{swe}; \xrightarrow{poT\text{-}}$ be the external component of $\xrightarrow{hb}$, which captures how synchronization propagates across different threads.

These definitions provides a clean decomposition of hb.

Lemma C.1 (Characterizing hb). $\xrightarrow{hb} = \xrightarrow{init} \cup \xrightarrow{hbe} \cup \xrightarrow{po}$

Proof. The inclusion of $\xrightarrow{init} \cup \xrightarrow{hbe} \cup \xrightarrow{po} \subseteq \xrightarrow{hb}$ is immediate.

For the converse direction. The following calculations are immediate.

$$\begin{array}{lll} \xrightarrow{init}; \xrightarrow{hb} & \subseteq & \xrightarrow{init} \\ \xrightarrow{poT\text{-}}; \xrightarrow{po\text{-}T} & \subseteq & \xrightarrow{poTT} \\ \xrightarrow{po}; \xrightarrow{hbe}; \xrightarrow{po} & \subseteq & \xrightarrow{hbe} \end{array}$$

Thus we are able to deduce that $\xrightarrow{hbe}; \xrightarrow{hbe} \subseteq \xrightarrow{hbe}$ as follows:

$$\begin{array}{ll} & \xrightarrow{hbe}; \xrightarrow{hbe} \\ = & \xrightarrow{po\text{-}T}; (\xrightarrow{swe}; \xrightarrow{poTT})^\star; \xrightarrow{swe}; \xrightarrow{poT\text{-}}; \\ & \qquad \xrightarrow{po\text{-}T}; (\xrightarrow{swe}; \xrightarrow{poTT})^\star; \xrightarrow{swe}; \xrightarrow{poT\text{-}} \\ \subseteq & \xrightarrow{po\text{-}T}; (\xrightarrow{swe}; \xrightarrow{poTT})^\star; \xrightarrow{swe}; \xrightarrow{poTT}; (\xrightarrow{swe}; \xrightarrow{poTT})^\star; \xrightarrow{swe}; \xrightarrow{poT\text{-}} \\ \subseteq & \xrightarrow{po\text{-}T}; (\xrightarrow{swe}; \xrightarrow{poTT})^\star; \xrightarrow{swe}; \xrightarrow{poT\text{-}} \\ = & \xrightarrow{hbe} \end{array}$$

Hence, $\xrightarrow{init} \cup \xrightarrow{hbe} \cup \xrightarrow{po}$ is transitive. The proof is completed by noting that $\xrightarrow{cwr} \cup \xrightarrow{xrw} \subseteq \xrightarrow{hbe} \cup \xrightarrow{po}$. □

They also provide an alternative characterization of consistency in the implementation model[1].

Let $\xrightarrow{wre} = \xrightarrow{lwr} \setminus \xrightarrow{po}$ be the external portion of the read-to-write relation, and $\xrightarrow{xrwe} = \xrightarrow{xrw} \setminus \xrightarrow{po}$ be the external portion of the transactional read-to-read relation.

Lemma C.2. *An execution is consistent in the implementation model iff the following hold.*

$$\begin{array}{r} (\xrightarrow{hbe} \cup \xrightarrow{poT\text{-}} \cup \xrightarrow{po\text{-}T} \cup \xrightarrow{poRW} \cup \xrightarrow{wre} \cup \xrightarrow{xrwe}) \text{ is acyclic.} \\ (\xrightarrow{init} \cup \xrightarrow{hbe} \cup \xrightarrow{poCon}); \xrightarrow{lww} \text{ is irreflexive.} \\ (\xrightarrow{init} \cup \xrightarrow{hbe} \cup \xrightarrow{poCon}); \xrightarrow{lrw} \text{ is irreflexive.} \end{array}$$

Proof. For causality, we need that $(\xrightarrow{hb} \cup \xrightarrow{lwr} \cup \xrightarrow{xrw})$ is acyclic. We deduce:

$$\begin{array}{ll} & \xrightarrow{hb} \cup \xrightarrow{lwr} \cup \xrightarrow{xrw} \text{ is acyclic.} \\ \Leftrightarrow & \xrightarrow{init} \cup \xrightarrow{hbe} \cup \xrightarrow{po} \cup \xrightarrow{lwr} \cup \xrightarrow{xrw} \text{ is acyclic.} \\ \Leftrightarrow & \xrightarrow{hbe} \cup \xrightarrow{po} \cup \xrightarrow{lwr} \cup \xrightarrow{xrw} \text{ is acyclic.} \\ \Leftrightarrow & \xrightarrow{hbe} \cup \xrightarrow{po} \cup \xrightarrow{wre} \cup \xrightarrow{xrwe} \text{ is acyclic.} \end{array}$$

The first step follows from Lemma C.1; the second since $\xrightarrow{init}$ is acyclic, and the last from definitions of $\xrightarrow{wre}$, $\xrightarrow{xrwe}$.

Consider a cycle in the last relation above. Without loss of generality, assume that every two adjacent elements of the cycle are in different threads. All the relations other than $\xrightarrow{wre}$ use transactional events. So, if we have two adjacent events $a \xrightarrow{po} b$, neither of which is transactional, the cycle

[1] We include $\xrightarrow{init}$ to be consistent with [9]. It can be removed since the initializing transaction has only one write per location; thus, initialization actions are not the target of any of our relations.

contains $c_1 \xrightarrow{\text{wre}} a \xrightarrow{\text{po}} b \xrightarrow{\text{wre}} c_2$. Thus, we deduce that $a \xrightarrow{\text{poRW}} b$.

In the last three items, we use Lemma C.1 for the alternative characterization of $\xrightarrow{\text{hb}}$. We can replace $\xrightarrow{\text{po}}$ by $\xrightarrow{\text{poCon}}$ by the following reasoning. If a ($\dashrightarrow^{\text{lww}} \cup \cdots\!\rightarrow^{\text{lrw}}$) b, then a, b access the same location and at least one is a write. □

The following lemma addresses the infrastructure needed for reordering transformations.

Lemma C.3. *Let σ, ρ be well-formed executions with the same events that agree on the $\xrightarrow{\text{init}}$, $\dashrightarrow^{\text{ww}}$, $\xrightarrow{\text{wr}}$, and $\overset{\text{tx}}{\sim}$ relations and satisfy:*

$$(\xrightarrow{\text{po-T}}_\sigma, \xrightarrow{\text{poT-}}_\sigma, \xrightarrow{\text{poTT}}_\sigma, \xrightarrow{\text{poRW}}_\sigma, \xrightarrow{\text{poCon}}_\sigma, \xrightarrow{\text{swe}}_\sigma) = (\xrightarrow{\text{po-T}}_\rho, \xrightarrow{\text{poT-}}_\rho, \xrightarrow{\text{poTT}}_\rho, \xrightarrow{\text{poRW}}_\rho, \xrightarrow{\text{poCon}}_\rho, \xrightarrow{\text{swe}}_\rho)$$

Then, σ is consistent iff ρ is consistent.

Proof. We first show that the happens-before relations of σ, ρ coincide. Since $\xrightarrow{\text{swe}}$ coincides for σ, ρ, $\xrightarrow{\text{hbe}}$ coincides for σ, ρ. Result is immediate using lemma C.1.

Since σ, ρ also agree on all the base relations $\xrightarrow{\text{init}}$, $\dashrightarrow^{\text{ww}}$, $\xrightarrow{\text{wr}}$, and $\overset{\text{tx}}{\sim}$, they also agree on all the derived lifted relations. Result follows. □

The following lemma addresses the infrastructure needed for roach-motel transformations.

Lemma C.4. *Let σ, ρ be well-formed executions with the same events that agree on the $\xrightarrow{\text{init}}$, $\dashrightarrow^{\text{ww}}$, $\xrightarrow{\text{wr}}$ and $\xrightarrow{\text{po}}$. Let the $\overset{\text{tx}}{\sim}$ relation of ρ be a superset of the $\overset{\text{tx}}{\sim}$ relation of σ.*

Then, if ρ is consistent, so is σ.

Proof. Since the $\overset{\text{tx}}{\sim}$ relation of σ is a subset of the $\overset{\text{tx}}{\sim}$ relation of σ, and σ, ρ agree on all the base relations $\xrightarrow{\text{init}}$, $\dashrightarrow^{\text{ww}}$, $\xrightarrow{\text{wr}}$, and $\xrightarrow{\text{po}}$, we deduce that all lifted relations of σ are a subset of the lifted relations of ρ and $\xrightarrow{\text{hb}}_\sigma \subseteq \xrightarrow{\text{hb}}_\rho$.

Consistency of σ follows from the consistency of ρ. □

The following lemma addresses the infrastructure needed for fusion transformations.

Lemma C.5. *Let ρ be a consistent, well-formed execution with transaction a in s. Let b be a new name. Let σ be derived from ρ by:*

- *introducing $\langle a{:}s\,C\rangle\langle b{:}s\,B\rangle$ between the begin and end of transaction a*
- *replacing the end (commit/abort) of a, if any, by an end (commit/abort) of b*

Then, σ is well-formed and consistent.

Proof. Well-formedness of σ follows from the well-formedness of ρ. WF_1–WF_8 are unaffected by the changes. Any violation of WF_9–WF_{11} in σ induces a violation of the same in ρ.

All orders in σ restricted to the actions from ρ are contained in the corresponding orders on ρ. Any simple cycle in any of the consistency criterion on σ induces a simple cycle in ρ with the new actions replaced by $\langle a{:}s\,\text{B}\rangle$. Thus, consistency of σ follows from consistency of ρ. □

The following lemma addresses the infrastructure needed for removing empty transactions.

Lemma C.6. *Let $\rho = \rho'\alpha\beta\rho''$ be a consistent, well-formed execution, where α is an action of s that is not part of any transaction.*

Let b be a new name. Let $\sigma = \rho'\alpha\langle b{:}s\,B\rangle\langle b{:}s\,C\beta\rangle\rho''$.

Then, σ is well-formed and consistent.

Proof. Well-formedness of σ follows immediately from the well-formedness of ρ. WF_1–WF_8 are unaffected by the changes. Any violation of WF_9–WF_{11} in σ induces a violation of the same in ρ.

The new actions in ρ only participate in the $\xrightarrow{\text{po}}$ order, where they have a unique predecessor and successor. All orders in σ restricted to the actions from ρ are contained in the corresponding orders on ρ. Any simple cycle in any of the consistency criterion on σ induces a simple cycle in ρ with the new actions replaced by $\alpha\beta$.

□

D Additional Examples

The next two examples discuss aborted transactions.

Example D.1 (Opaque writes)**.** Final outcome $r = 1$ is not permitted in the program below.

$$\text{atomic}_a\,\{\,x := 1;\ \text{abort}\,\}\ \|\ \text{atomic}_b\,\{\,r := x\,\}$$

This is trivial to justify by well-formedness (condition 7) since $\xrightarrow{\text{wr}}$ cannot originate from an aborted transaction.

Example D.2 (Race-free speculation)**.** The only permitted final outcome is $r = 2$.

$$\text{atomic}_a\,\{\,x{+}{+};\ y{+}{+}\,\}\ \|\ \text{atomic}_b\{\text{if } x \neq y \text{ then } \{z := 1;\ \text{abort}\}\}\ \|\ z := 2;\ r := z$$

Since the guard of transaction b will never hold the program is race free, and hence it will never execute $z{:=}1$. This means that there is no danger that the abort will undo the nontransactional write to z. In particular, for every execution $\langle \text{W}z2\rangle$ obscures the read of z in the third thread.

Example D.3 (Dirty reads)**.** Final result $x = 0$ and $y = 1$ is forbidden.

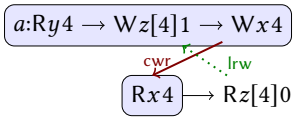
$$\text{atomic}_a\,\{\text{if } !y \text{ then } x := 1;\ \text{abort}\};\ \text{atomic}_b\,\{\text{if } !y \text{ then } x := 1\}\ \|\ \text{if } x = 1 \text{ then } y := 1$$

The result would be possible if the second thread observes the write of x in transaction a, then updates y. Since a rolls back, it will restore x's value back to 0, causing transaction b to skip over the update to x on re-execution. However, in our model such an execution is not possible since non-transactional events cannot read from live or aborted transactions.

Example D.4 (No overlapped writes)**.** Final result $r = 0$ is forbidden in the program below, where z is an array. The result would be possible if transaction a initializes $z[y]$ and then publishes it by writing it to shared *volatile* variable x.

Since lazy version copies cached values in any order, the second thread may see the update to x before it sees the update to $z[y]$. In our model, this results in the execution below.

$\text{atomic}_a\ \{y := 4;\ z[y] := 1;\ x := 4\}$
$\|\ r := 1;\ \text{atomic}\ \{\ q := x\ \};$
$\quad \text{if}\ q \neq 0\ \text{then}\ r := z[q]$



Since we model volatile accesses as a singleton committed transaction, we obtain an edge $\xrightarrow{\text{cwr}}$ to the read of x in the second thread, which violates axiom (Observation).