Privatization-Safe Transactional Memories

Artem Khyzha
Tel Aviv University, Tel Aviv, Israel

Hagit Attiya
Technion – Israel Institute of Technology, Haifa, Israel

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain

Abstract

Transactional memory (TM) facilitates the development of concurrent applications by letting the programmer designate certain code blocks as atomic. Programmers using a TM often would like to access the same data both inside and outside transactions, and would prefer their programs to have a *strongly atomic* semantics, which allows transactions to be viewed as executing atomically with respect to non-transactional accesses. Since guaranteeing such semantics for arbitrary programs is prohibitively expensive, researchers have suggested guaranteeing it only for certain *data-race free (DRF)* programs, particularly those that follow the *privatization* idiom: from some point on, threads agree that a given object can be accessed non-transactionally.

In this paper we show that a variant of *Transactional DRF (TDRF)* by Dalessandro et al. is appropriate for a class of *privatization-safe* TMs, which allow using privatization idioms. We prove that, if such a TM satisfies a condition we call *privatization-safe opacity* and a program using the TM is TDRF under strongly atomic semantics, then the program indeed has such semantics. We also present a method for proving privatization-safe opacity that reduces proving this generalization to proving the usual opacity, and apply the method to a TM based on two-phase locking and a privatization-safe version of TL2. Finally, we establish the inherent cost of privatization-safety: we prove that a TM cannot be progressive and have invisible reads if it guarantees strongly atomic semantics for TDRF programs.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Program semantics; Software and its engineering → Software verification

Keywords and phrases Transactional memory, privatization, observational refinement

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.24

Related Version An extended version is available at `https://arxiv.org/abs/1908.03179`.

Funding This research was funded in part by the Israel Science Foundation (grants 2005/17, 1749/14 and 380/18) and the European Research Council (Starting Grant RACCOON).

1 Introduction

Transactional memory (TM) facilitates the development of concurrent applications by letting the programmer designate certain code blocks as *atomic* [23]. TM allows developing a program and reasoning about its correctness as if each atomic block executes as a *transaction* – atomically and without interleaving with other blocks – even though in reality the blocks can be executed concurrently. A TM can be implemented in hardware [24, 29], software [34] or a combination of both [13, 28].

Often programmers using a TM would like to access the same data both inside and outside transactions. This may be desirable to avoid performance overheads of transactional accesses, to support legacy code, or for explicit memory deallocation. One typical pattern is *privatization* [31, 35], illustrated in Figure 1. There the `atomic` blocks return a value signifying whether the transaction committed or aborted. In the program, an object `x` is

```
{ priv = false ∧ x = 0 }
l1 = atomic {                 ||  atomic {
  priv = true; } // T1        ||    if (!priv)
if (l1 == committed)          ||      x = 42;
  x = 1; // n                 ||  } // T2
{ l1 = committed ⟹ x = 1 }
```

Figure 1 Privatization.

```
{ priv = true ∧ x = l = 0 }
x = 42; // n                  ||  l2 = atomic {
l1 = atomic {                 ||    if (!priv)
  priv = false;               ||      l = x;
} // T1                       ||  } // T2
{ l2 = committed ∧ l ≠ 0 ⟹ l = 42 }
```

Figure 2 Publication.

```
{ x = y = 0 }
atomic {      ||
  x = 1;      ||  l1 = x; // n1
  y = 2;      ||  l2 = y; // n2
} // T        ||
{ l1 = 1 ⟹ l2 = 2 }
```

Figure 3 Data race.

guarded by a flag `priv`, showing whether the object should be accessed transactionally (`false`) or non-transactionally (`true`). The left-hand-side thread first tries to set the flag inside transaction T_1, thereby *privatizing* `x`. If successful, it then accesses `x` non-transactionally. A concurrent transaction T_2 in the right-hand-side thread checks the flag `priv` prior to accessing `x`, to avoid simultaneous transactional and non-transactional access to the object. We expect the postcondition shown to hold: if privatization is successful, at the end of the program `x` should store 1, not 42. The opposite idiom is *publication*, illustrated in Figure 2. The left-hand-side thread writes to `x` non-transactionally and then clears the flag `priv` inside transaction T_1, thereby *publishing* `x`. The right-hand-side thread tests the flag inside transaction T_2, and if it is cleared, reads `x`. Again, we expect the postcondition to hold: if the right-hand-side thread sees the write to the flag, it should also see the write to `x`. The two idioms can be combined: the programmer may privatize an object, then access it non-transactionally, and finally publish it back for transactional access.

Ideally, programmers mixing transactional and non-transactional accesses to objects would like their programs to have *strongly atomic semantics* [8], where transactions can be viewed as executing atomically also with respect to non-transactional accesses, i.e., without interleaving with them. This is equivalent to considering every non-transactional access as a single-instruction transaction. For example, the program in Figure 3 under strongly atomic semantics can only produce executions where each of the non-transactional accesses n_1 and n_2 executes either before or after the transaction T, so that the postcondition in Figure 3 always holds. Unfortunately, providing such semantics in software requires instrumenting non-transactional accesses with additional instructions for maintaining TM metadata [19]. This undermines scalability and makes it difficult to reuse legacy code. Since most existing TMs are either software-based or rely on a software fall-back, they do not perform such instrumentation and, hence, provide weaker atomicity guarantees. For example, they may allow the program in Figure 3 to execute non-transactional accesses n_1 and n_2 between transactional writes to `x` and `y` and, thus, observe an intermediate state of the transaction, e.g., $\mathtt{x} = 1$ and $\mathtt{y} = 0$, violating the postcondition in Figure 3.

Researchers have suggested resolving the tension between strong TM semantics and performance by guaranteeing strongly atomic semantics only to *data-race free (DRF)* programs – informally, programs without concurrent transactional and non-transactional accesses to the same data [4, 5, 10, 11, 31, 33, 35]. For example, we do not have to guarantee strongly atomic semantics for the program in Figure 3, which has such concurrent accesses to `x` and `y`. On the other hand, the programs in Figure 1 and Figure 2 should be guaranteed strongly atomic semantics, since at any point of time, an object is accessed either only transactionally or only non-transactionally. Despite the intuitive simplicity of this idea, coming up with a precise DRF definition is nontrivial: early on there were multiple competing proposals for the notion of DRF, and it was unclear how to select among them [4, 10, 11, 25, 30]. To address this, we have recently formalized the requirements on an appropriate notion of DRF using observational refinement [27]: a TM needs to guarantee that, if a program is DRF under the strongly atomic semantics (formalized as *transactional sequential consistency* [11]), then all its executions are observationally equivalent to strongly atomic ones. This *Fundamental Property* allows the programmer to never reason about weakly atomic semantics at all, even when checking DRF.

Different TMs have different requirements on mixing transactional and non-transactional accesses needed to validate the Fundamental Property. *Privatization-safe* TMs, such as lock-based TMs [15, 21] and NOrec [12], allow the programmer to ensure the absence of concurrent transactional and non-transactional accesses by synchronizing them using transactional operations. Then the program in Figure 1, which synchronizes accesses to `x` using `priv`, is guaranteed strongly atomic semantics as is. *Privatization-unsafe* TMs, such as TL2 [14] and TinySTM [16], require the programmer to insert additional synchronization, e.g., via *transactional fences* [31, 35], which block until all the transactions that were active when the fence was invoked complete. For example, such TMs do not guarantee strongly atomic semantics to the program in Figure 1 unless the transaction T_1 is immediately followed by a transactional fence. This is because TMs such as TL2 execute transactions optimistically, flushing their writes to memory only on commit. Then, in the absence of a fence, the transaction T_1 can privatize `x` and n can modify it after T_2 started committing, but before its write to `x` reached the memory, so that T_2's write subsequently overwrites n's write and violates the postcondition. TMs that make transactional updates in-place and undo them on abort are subject to a similar problem.

Privatization-safe TMs provide a simpler programming model, since they do not require the programmer to select where to place fences. However, the programmer still needs to avoid programs of the kind shown in Figure 3, which would lead the TM to violate strong atomicity. In this paper we show that a variant of *transactional DRF (TDRF)* previously proposed by Dalessandro et al. [11] is appropriate to formalize the programmer's obligations. To this end, we show that this variant of TDRF validates the Fundamental Property, provided the TM satisfies a generalization of *opacity* [20, 21], which we call *privatization-safe opacity.* To formulate this kind of opacity, we generalize TDRF to arbitrary TM histories, not just strongly atomic ones. These results complement our previous proposal of DRF for privatization-unsafe TMs, which considers a more low-level programming model requiring fence placements [27].

We furthermore present a method for proving privatization-safe opacity and apply it to a TM based on two-phase locking [21] and a privatization-safe version of TL2 [14] that executes a fence at the end of each transaction. A key feature of our method is that it reduces proving privatization-safe opacity to proving the ordinary opacity of the TM assuming no mixed transactional/non-transactional accesses. This allows us to reuse the previous proofs of opacity of the two-phase locking TM [21] and TL2 [27].

Finally, our framework allows proving an interesting result about the inherent cost of privatization-safety. We prove that a TM that provides strongly atomic semantics to TDRF programs cannot be *progressive* and have *invisible reads*: it cannot ensure that transactions always complete when running solo and also that transactions reading objects do not prevent transactions writing to them from committing. This result significantly simplifies and strengthens a lower bound by Attiya and Hillel [7], which did not use a formal DRF notion.

2 Programming Language and Strongly Atomic Semantics

Language syntax. We formalize our results for a simple programming language with mixed transactional and non-transactional accesses. A *program* $P = C_1 \parallel \ldots \parallel C_N$ in our language is a parallel composition of *commands* C_t executed by different *threads* $t \in \mathsf{ThreadID} =$

$\{1, \dots, N\}$. Every thread $t \in \mathsf{ThreadID}$ has a set of *local variables* $l \in \mathsf{LVar}_t$, which only it can access; for simplicity, we assume that these are integer-valued. Threads have access to a *transactional memory* (*TM*), which manages a fixed collection of *shared register objects* $x \in \mathsf{Reg}$. The syntax of commands $C \in \mathsf{Com}$ is as follows:

$$\begin{aligned} C ::= {} & c \mid C\,;C \mid \ \mathtt{if}\,(b)\ \mathtt{then}\ C\ \mathtt{else}\ C \mid \ \mathtt{while}\,(b)\ \mathtt{do}\ C \\ & \mid l = \mathtt{atomic}\,\{C\} \mid l = x.\mathtt{read}() \ \mid x.\mathtt{write}(e) \end{aligned}$$

where b and e denote Boolean, respectively, integer *expressions* over local variables and constants. The language includes *primitive commands* $c \in \mathsf{PCom}$, which operate on local variables, and standard control-flow constructs. An *atomic block* $l = \mathtt{atomic}\,\{C\}$ executes C as a *transaction*, which the TM can *commit* or *abort*. The system's decision is returned in the local variable l, which receives a distinguished value committed or aborted. We do not allow programs to abort a transaction explicitly and forbid nested atomic blocks. Threads can invoke two methods on a register x: $x.\mathtt{read}()$ returns the current value of x, and $x.\mathtt{write}(e)$ sets it to e. These methods may be invoked both *inside* and *outside* atomic blocks.

Model of computations. The semantics of our programming language is defined in terms of *traces* – certain finite sequences of *actions*, each describing a single computation step (in this paper we consider only finite computations). Let ActionId be a set of *action identifiers*. Actions are of two kinds. A *primitive action* denotes the execution of a primitive command and is of the form (a, t, c), where $a \in \mathsf{ActionId}$, $t \in \mathsf{ThreadID}$ and $c \in \mathsf{PCom}$. An *interface action* has one of the following forms (where $x \in \mathsf{Reg}$ and $v \in \mathbb{Z}$):

Request actions	Matching response actions
$(a, t, \mathsf{begintx})$	$(a, t, \mathsf{ok}) \mid (a, t, \mathsf{aborted})$
$(a, t, \mathsf{trycommit})$	$(a, t, \mathsf{committed}) \mid (a, t, \mathsf{aborted})$
$(a, t, \mathsf{write}(x, v))$	$(a, t, \mathsf{ret}(\bot)) \mid (a, t, \mathsf{aborted})$
$(a, t, \mathsf{read}(x))$	$(a, t, \mathsf{ret}(v)) \mid (a, t, \mathsf{aborted})$

Interface actions usually denote the control flow of a thread t crossing the boundary between the program and the TM: *request* actions correspond to the control being transferred from the former to the latter, and *response* actions, the other way around. A begintx action is generated upon entering an `atomic` block, and a trycommit action when a transaction tries to commit upon exiting an `atomic` block. The request actions $\mathsf{write}(x, v)$ and $\mathsf{read}(x)$ denote invocations of the `write`, respectively, `read` methods of register x; a `write` action is annotated with the value v written. The response actions $\mathsf{ret}(\bot)$ and $\mathsf{ret}(v)$ denote the return from invocations of `write`, respectively, `read` methods of a register; the latter is annotated with the value v read. The TM may abort a transaction at any point when it is in control; this is recorded by an aborted response action. To simplify notation, we reuse the interface actions for reads and writes to denote accesses outside transactions.

A *trace* τ is a finite sequence of actions satisfying the expected well-formedness conditions, e.g., that request and response actions are properly matched, and so are actions denoting the beginning and the end of transactions (we defer the formal definition to [26, §A]). A *transaction* T is a nonempty trace such that it contains actions by the same thread, begins with a begintx action and only its last action can be a committed or an aborted action. A transaction T is: *committed* if it ends with a committed action, *aborted* if it ends with aborted, *commit-pending* if it ends with trycommit, and *live*, in all other cases. A transaction T is in a trace τ if T is a subsequence of τ and no longer transaction is. We refer to interface actions in a trace outside of a transaction as *non-transactional actions*. We call a matching request/response pair of a read or a write a *non-transactional access* (ranged over by n).

A *history* is a trace containing only interface actions (thus, omitting all accesses to local variables); we use H, S to range over histories, and $H(i)$ to refer to the i-th action in H. We also use $\mathsf{history}(\tau)$ to denote a projection of a trace to interface actions. Since histories fully capture the possible interactions between a TM and a client program, we often conflate the notion of a TM and the set of histories it produces. Hence, a *transactional memory* $\mathcal{H}$ is a prefix-closed set of histories. We assume that a TM always allows a client program to execute a request and, hence, require $\mathcal{H}$ to be closed under appending any request action to its histories, provided that the latter remain well-formed. Note that histories include actions corresponding to non-transactional accesses, even though these may not be directly managed by the TM implementation. This is needed to account for changes to registers performed by such actions when defining the TM semantics: e.g., in the case when a register is privatized, modified non-transactionally and then published back for transactional access. Of course, a well-formed TM semantics should not impose restrictions on the placement of non-transactional actions, since these are under the control of the program.

Strongly atomic semantics. The *semantics* of a program P is given by the set $[\![P]\!](\mathcal{H})$ of traces it produces when executed with a TM $\mathcal{H}$. Its formal definition follows the intuitive meaning of commands, and we defer it to [26, §A]. Our semantics assumes that the underlying memory is sequentially consistent, which allows us to focus on the key issues specific to TM (we leave handling weak memory for future work, discussed in §9). We use the semantics instantiated with one particular TM to define the *strongly atomic* semantics of programs [8], which is equivalent to transactional sequential consistency [11]. Following [6], we use an *atomic* TM $\mathcal{H}_{\mathsf{atomic}}$ for this purpose: the strongly atomic semantics of a program P is given by the set of traces $[\![P]\!](\mathcal{H}_{\mathsf{atomic}})$. The TM $\mathcal{H}_{\mathsf{atomic}}$ contains only histories that are *non-interleaved*, i.e., where actions by one transaction do not overlap with actions of another transaction or of non-transactional accesses. Out of such histories, $\mathcal{H}_{\mathsf{atomic}}$ contains only histories following the intuitive atomic semantics of transactions: every response action of a $\mathsf{read}(x)$ returns the value v in the last preceding $\mathsf{write}(x, v)$ action that is not located in an aborted or live transaction different from the one of the read; if there is no such write, the read returns the initial value v_{init}. We defer a formal definition of $\mathcal{H}_{\mathsf{atomic}}$ to [26, §A].

3 Transactional Data-Race Freedom

We now formalize in our framework a variant of *transactional data-race freedom (TDRF)* of Dalessandro et al. [11]. According to this notion, a data race happens between a pair of *conflicting* actions, as defined below.

▶ **Definition 1.** *A non-transactional request action α and a transactional request action α'* conflict *if α and α' are executed by different threads, they are read or write actions on the same register, and at least one of them is a write.*

As is standard, we formalize when conflicting actions form a data race using a *happens-before* relation $\mathsf{hb}(H)$ on actions in a history H. We first define the *execution order* of H as follows: $\alpha <_H \alpha'$ iff for some i and j, $\alpha = H(i)$, $\alpha' = H(j)$ and $i < j$.

▶ **Definition 2.** *The* happens-before *relation of a history $H \in \mathcal{H}_{\mathsf{atomic}}$ is* $\mathsf{hb}(H) \triangleq (\mathsf{po}(H) \cup \mathsf{ef}(H) \cup \mathsf{cl}(H))^+$*, where*

- per-thread order $\mathsf{po}(H)$*: $\alpha <_{\mathsf{po}(H)} \alpha'$ iff $\alpha <_H \alpha'$ and α, α' are by the same thread;*
- effect order $\mathsf{ef}(H)$*: $\alpha <_{\mathsf{ef}(H)} \alpha'$ iff $\alpha <_H \alpha'$ and α, α' are by different transactions;*
- client order $\mathsf{cl}(H)$*: $\alpha <_{\mathsf{cl}(H)} \alpha'$ iff $\alpha <_H \alpha'$ and α, α' are non-transactional in H.*

▶ **Definition 3.** *A history* $H \in \mathcal{H}_{\mathsf{atomic}}$ *is* transactional data-race free*, written* $\mathsf{TDRF}(H)$*, if every pair of conflicting actions in it is ordered by* $\mathsf{hb}(H)$ *one way or another. A program* P *is* transactional data-race free*, written* $\mathsf{TDRF}(P)$*, if* $\forall \tau \in [\![P]\!](\mathcal{H}_{\mathsf{atomic}}).\, \mathsf{TDRF}(\mathsf{history}(\tau))$*.*

Components of happens-before used to define TDRF describe various forms of synchronization available in our programming language. First, actions by the same thread cannot be concurrent and thus we let $\mathsf{po}(H) \subseteq \mathsf{hb}(H)$. Second, privatization-safe TMs provide synchronization between transactions, which follows their order in non-interleaved histories of an atomic TM considered in the definition of TDRF on programs. Thus, we let $\mathsf{ef}(H) \subseteq \mathsf{hb}(H)$. Finally, we let $\mathsf{cl}(H) \subseteq \mathsf{hb}(H)$, because in this paper we assume a sequentially consistent memory model and, hence, do not consider pairs of conflicting non-transactional accesses as races. This is the key difference between our variant of TDRF and the original definition by Dalessandro et al. [11], which does not include the client order into happens-before. Our variant of TDRF imposes fewer obligations on the programmer: as we show by establishing the Fundamental Property for our variant of TDRF (§5), under sequentially consistent memory races on non-transactional accesses are harmless for privatization-safety.

To illustrate the TDRF definition, we show that the program in Figure 1 is TDRF by considering the histories it produces with the atomic TM (the program in Figure 2 can be shown TDRF analogously). The possible conflicts are between the accesses to `x` in n and T_2. For a conflict to occur, T_2 has to read `false` from `priv`; then T_2 has to execute before T_1, yielding a history of the form $T_2 T_1 n$. In this history T_2 precedes T_1 in the effect order and T_1 precedes n in the per-thread order, meaning that $\mathsf{hb}(H)$ orders the conflict between T_2 and n. Similarly, in [26, §B] we show that programs following a *proxy privatization* pattern [37], where one thread privatizes an object for another thread, are also TDRF. On the other hand, the program in Figure 3 is not TDRF, since in histories it produces with the atomic TM, the happens-before never relates T with n_1 and n_2. Finally, the inclusion of $\mathsf{cl}(H) \subseteq \mathsf{hb}(H)$ allows us to consider DRF those programs that privatize an object by agreeing on its status outside transactions ("partitioning by consensus" in [35]); we provide an example in [26, §B].

4 Privatization-Safe Opacity

We now present our first contribution – a generalization of *opacity* of a TM $\mathcal{H}$ [20, 21] that guarantees that the TM provides strongly atomic semantics to TDRF programs. We call this generalization *privatization-safe opacity*. Its definition requires that a history H of a TM $\mathcal{H}$ can be matched by a history S of the atomic TM $\mathcal{H}_{\mathsf{atomic}}$ that "looks similar" to H from the perspective of the program. The similarity is formalized by a relation $H \sqsubseteq S$, which requires S to be a permutation of H preserving its per-thread and client orders.

▶ **Definition 4.** *A history* H_1 *corresponds to a history* H_2*, written* $H_1 \sqsubseteq H_2$*, if there is a bijection* $\theta : \{1, \ldots, |H_1|\} \to \{1, \ldots, |H_2|\}$ *such that* $\forall i.\, H_1(i) = H_2(\theta(i))$ *and*

$$\forall i, j.\, i < j \wedge H_1(i) <_{\mathsf{po}(H_1) \cup \mathsf{cl}(H_1)} H_2(j) \implies \theta(i) < \theta(j).$$

The above relation differs in several ways from the one used to define the ordinary opacity. First, unlike in the ordinary opacity, our histories include non-transactional actions, because these can affect the behavior of the TM. Second, instead of preserving $\mathsf{cl}(H_1)$ in Definition 4, the ordinary opacity requires preserving the following *real-time order* $\mathsf{rt}(H_1)$ on actions: $\alpha <_{\mathsf{rt}(H)} \alpha'$ iff $\alpha \in \{(\_, \_, \mathsf{committed}), (\_, \_, \mathsf{aborted})\}$, $\alpha' = (\_, \_, \mathsf{begintx})$ and $\alpha <_H \alpha'$. This orders non-overlapping transactions, with the duration of a transaction determined by the interval from its begintx action to the corresponding committed or aborted action (or to the end of the history if there is none). However, preserving real-time order is unnecessary if all means of communication between program threads are reflected in histories [17].

We next lift privatization-safe opacity to TMs. A straightforward definition, mirroring the ordinary opacity, would require any history of the TM $\mathcal{H}$ to have a matching history of the atomic TM $\mathcal{H}_{\mathsf{atomic}}$. However, such a requirement would be too strong for our setting: since the TM has no control over non-transactional actions of its clients, histories in $\mathcal{H}$ may be produced by racy programs, and we do not want to require the TM to guarantee strong atomicity in such cases. For example, even though a simple TM based on a single global lock is privatization-safe, it has a history produced by the program from Figure 3 that does not have a matching history of $\mathcal{H}_{\mathsf{atomic}}$ (§1). Hence, our definition of privatization-safe opacity requires only histories produced by TDRF programs to have justifications in $\mathcal{H}_{\mathsf{atomic}}$. To express this restriction, we generalize data-race freedom to be defined over an arbitrary concurrent history H, not just one produced by $\mathcal{H}_{\mathsf{atomic}}$. The new DRF requires that every history of the atomic TM matching H according to the opacity relation be TDRF.

▶ **Definition 5.** *A history $H \in \mathcal{H}$ is* concurrent data-race free*, written* $\mathsf{CDRF}(H)$*, if* $\forall S \in \mathcal{H}_{\mathsf{atomic}}.\, H \sqsubseteq S \implies \mathsf{TDRF}(S)$*. Let* $\mathcal{H}|_{\mathsf{CDRF}} = \{H \in \mathcal{H} \mid \mathsf{CDRF}(H)\}$*. A program P is concurrent data-race free with a TM $\mathcal{H}$, written* $\mathsf{CDRF}(P, \mathcal{H})$*, if* $\forall \tau \in [\![P]\!](\mathcal{H}).\, \mathsf{CDRF}(\mathsf{history}(\tau))$*.*

▶ **Definition 6.** *A TM $\mathcal{H}$ is* privatization-safe opaque*, written* $\mathcal{H}|_{\mathsf{CDRF}} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$*, if for every history $H \in \mathcal{H}|_{\mathsf{CDRF}}$ there exists a history $S \in \mathcal{H}_{\mathsf{atomic}}$ such that $H \sqsubseteq S$ holds.*

The following lemma (proved in [26, §C]) justifies using CDRF as a generalization of TDRF to concurrent histories by establishing that TDRF programs indeed produce CDRF histories.

▶ **Lemma 7.** *For every program P and a TM system $\mathcal{H}$,* $\mathsf{TDRF}(P)$ *implies* $\mathsf{CDRF}(P, \mathcal{H})$*.*

5 The Fundamental Property

We next formalize the Fundamental Property of TDRF using *observational refinement* [6]: if a program is TDRF under the atomic TM $\mathcal{H}_{\mathsf{atomic}}$, then any trace of the program under a privatization-safe opaque TM $\mathcal{H}$ has an *observationally equivalent* trace under $\mathcal{H}_{\mathsf{atomic}}$.

▶ **Definition 8.** *Traces τ and τ' are* observationally equivalent*, denoted by $\tau \sim \tau'$, if $\forall t.\, \tau|_t = \tau'|_t$ and $\tau|_{\mathsf{nontx}} = \tau'|_{\mathsf{nontx}}$, where $\tau|_{\mathsf{nontx}}$ denotes the subsequence of τ containing all actions from non-transactional accesses.*

Equivalent traces are considered indistinguishable to the user. In particular, the sequences of non-transactional accesses in equivalent traces (which usually include all I/O) satisfy the same linear-time temporal properties. We lift the equivalence to sets of traces as follows.

▶ **Definition 9.** *A set of traces $\mathcal{T}$* observationally refines *a set of traces $\mathcal{T}'$, written $\mathcal{T} \preceq \mathcal{T}'$, if $\forall \tau \in \mathcal{T}.\, \exists \tau' \in \mathcal{T}'.\, \tau \sim \tau'$.*

▶ **Theorem 10** (Fundamental Property)**.** *If $\mathcal{H}$ is a TM such that $\mathcal{H}|_{\mathsf{CDRF}} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$ and P is a program such that* $\mathsf{TDRF}(P)$*, then $[\![P]\!](\mathcal{H}) \preceq [\![P]\!](\mathcal{H}_{\mathsf{atomic}})$.*

Theorem 10 establishes a contract between the programmer and the TM implementors. The TM implementor has to ensure privatization-safe opacity of the TM assuming the program is DRF: $\mathcal{H}|_{\mathsf{CDRF}} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$. The programmer has to ensure the DRF of the program under strongly atomic semantics: $\mathsf{TDRF}(P)$. This contract lets the programmer check properties of a program assuming strongly atomic semantics ($[\![P]\!](\mathcal{H}_{\mathsf{atomic}})$) and get the guarantee that the properties hold when the program uses the actual TM implementation ($[\![P]\!](\mathcal{H})$). Theorem 10 follows from Lemma 7 and the next lemma, which is an adaptation of a result from [6].

▶ **Lemma 11.** *If $\mathcal{H}$ is a TM such that $\mathcal{H}|_{\mathsf{CDRF}} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$, then $\forall P.\, \mathsf{CDRF}(P, \mathcal{H}) \implies [\![P]\!](\mathcal{H}) \preceq [\![P]\!](\mathcal{H}_{\mathsf{atomic}})$.*

6 Proving Privatization-Safe Opacity

We now develop a method that reduces proving privatization-safe opacity ($\mathcal{H}|_{\mathsf{CDRF}} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$) to proving the ordinary opacity. The method builds on a *graph characterization* of opacity by Guerraoui and Kapalka [21], which was proposed for proving opacity of TMs that do not allow mixed transactional/non-transactional accesses to the same data. The characterization allows checking opacity of a history H by checking two properties: *consistency* of the history, denoted $\mathsf{cons}(H)$, and the acyclicity of a certain *opacity graph*, which we define in the following. Consistency is a basic well-formedness property of a history ensuring the following. If a transaction T in H reads a value of a register x and writes to it before, then T reads the latest value it writes. If T reads a value of x and does not write to it before, then it reads some value written non-transactionally or by a committed or commit-pending transaction (or the initial value, when everything else fails). Consistency also ensures that only the last write to x by a transaction is read from. We define consistency formally in [26, §D] and focus here on defining opacity graphs.

The vertexes in these graphs include transactions and non-transactional accesses in H. The intention of the vis predicate below is to mark those vertexes that have taken effect, including commit-pending transactions of this kind. The other components, intuitively, constrain the order in which the vertexes should appear in the atomic history.

▶ **Definition 12.** *The* opacity graph *of a history H is a tuple $G = (\mathcal{V}, \mathsf{vis}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{PO}, \mathsf{CL})$, where:*

- *$\mathcal{V}$ is the set of graph vertexes, i.e., all transactions and non-transactional accesses from H (ranged over by ν).*
- *$\mathsf{vis} \subseteq \mathcal{V}$ is a* visibility *predicate, which holds of all non-transactional accesses and committed transactions and does not hold of all aborted and live transactions.*
- *$\mathsf{WR} : \mathsf{Reg} \to 2^{\mathcal{V}\times\mathcal{V}}$ specifies per-register* read-dependency *relations on vertexes, such that*
  - *For each read dependency $\nu \xrightarrow{\mathsf{WR}_x} \nu'$, we have that $\nu \neq \nu'$, ν contains $(\_, \_, \mathsf{write}(x, v))$, and ν' contains a request $(\_, \_, \mathsf{read}(x))$ and a matching response $(\_, \_, \mathsf{ret}(v))$.*
  - *Each vertex that reads x has at most one corresponding read dependency: $\forall \nu, \nu', \nu'', x.\, \nu \xrightarrow{\mathsf{WR}_x} \nu' \wedge \nu'' \xrightarrow{\mathsf{WR}_x} \nu \implies \nu = \nu''$.*
  - *Each vertex that is read from is visible: $\forall \nu, x.\, \nu \xrightarrow{\mathsf{WR}_x} \_ \implies \mathsf{vis}(\nu)$.*

  Informally, $\nu \xrightarrow{\mathsf{WR}_x} \nu'$ means that ν' reads what ν wrote to x.
- *$\mathsf{WW} : \mathsf{Reg} \to 2^{\mathcal{V}\times\mathcal{V}}$ specifies per-register* write-dependency *relations, such that for each $x \in \mathsf{Reg}$, WW_x is an irreflexive total order on $\{\nu \in \mathcal{V} \mid \mathsf{vis}(\nu) \wedge (\_, \_, \mathsf{write}(x, \_)) \in \nu\}$. Informally, $\nu \xrightarrow{\mathsf{WW}_x} \nu'$ means that ν' overwrites what ν wrote to x.*
- *$\mathsf{RW} \in \mathsf{Reg} \to 2^{\mathcal{V}\times\mathcal{V}}$ specifies per-register* anti-dependency *relations:*

$$\nu \xrightarrow{\mathsf{RW}_x} \nu' \iff \nu \neq \nu' \wedge ((\exists \nu''.\, \nu'' \xrightarrow{\mathsf{WW}_x} \nu' \wedge \nu'' \xrightarrow{\mathsf{WR}_x} \nu) \vee (\mathsf{vis}(\nu') \wedge (\_, \_, \mathsf{write}(x, \_)) \in \nu' \wedge (\_, \_, \mathsf{ret}(x, v_{\mathsf{init}})) \in \nu)).$$

  Informally, $\nu \xrightarrow{\mathsf{RW}_x} \nu'$ means that ν' overwrites the write to x that ν previously read (the initial value of x is overwritten by any write to x).
- *$\mathsf{PO}, \mathsf{CL} \in 2^{\mathcal{V}\times\mathcal{V}}$ are the per-thread and client orders lifted to pairs of graph vertexes: e.g., $\nu \xrightarrow{\mathsf{PO}(H)} \nu' \iff \exists \alpha \in \nu, \alpha' \in \nu'.\, \alpha <_{\mathsf{po}(H)} \alpha'$.*

We let $\mathsf{Graph}(H)$ denote the set of all opacity graphs of H. We say that a graph G is *acyclic*, written $\mathsf{acyclic}(G)$, if its edges do not form a directed cycle. We also refer to histories resulting from topological sortings of vertexes in a graph G as its *linearizations* and denote their set by $\mathsf{lins}(G)$. The next lemma shows that we can check privatization-safe opacity of a history by checking its consistency and the acyclicity of its opacity graph, with any linearization of the graph yielding a matching atomic history.

▶ **Lemma 13.** $\forall H. (\mathsf{cons}(H) \wedge \exists G \in \mathsf{Graph}(H). \mathsf{acyclic}(G)) \implies \mathsf{lins}(G) \subseteq \mathcal{H}_{\mathsf{atomic}}$.

The lemma is proven analogously to Lemma 6.4 in [27, §B.2]. It implies the following theorem, which gives a criterion for the privatization-safe opacity of a TM $\mathcal{H}$.

▶ **Theorem 14.** *$\mathcal{H} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$ holds if $\forall H \in \mathcal{H}. \mathsf{cons}(H) \wedge \exists G \in \mathsf{Graph}(H). \mathsf{acyclic}(G)$.*

In comparison to the graph characterization of the ordinary opacity [21], ours is more complex: the graph includes non-transactional accesses and the acyclicity check has to take into account paths involving them. We now formulate lemmas that simplify reasoning about non-transactional operations: they allow proving the privatization-safe opacity of a TM using Theorem 14 with only small adjustments to a proof of its ordinary opacity using graph characterization. The latter characterization includes only transactions as nodes of the graph, but additionally considers paths including the lifting of the real-time order from §4 to transactions: for a history H, we let $\mathsf{RT}(H)$ be the relation between transactions in H such that $T <_{\mathsf{RT}(H)} T'$ iff for some $\alpha \in T$ and $\alpha' \in T$ we have $\alpha <_{\mathsf{rt}(H)} \alpha'$. We also let DEP denote any edge in a given graph G, and we let txDEP denote an edge between two transactions.

The following lemma exploits CDRF to show that, for every path between two transactions in an acyclic opacity graph, there is another path replacing edges involving non-transactional accesses by real-time order edges or transactional dependencies.

▶ **Lemma 15.** *Consider an acyclic opacity graph $G = (\mathcal{V}, \mathsf{vis}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{PO}, \mathsf{CL})$ of a consistent CDRF history H. For any two transactions T and T', if $T \xrightarrow{\mathsf{DEP}}{}^{*} T'$, then $T \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}}{}^{*} T'$.*

The next lemma exploits CDRF to show that, for every path between a transaction and a non-transactional access in an acyclic opacity graph, there is another path where per-thread order is the only kind of an edge between transactions and non-transactional accesses.

▶ **Lemma 16.** *Consider an acyclic opacity graph $G = (\mathcal{V}, \mathsf{vis}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{PO}, \mathsf{CL})$ of a CDRF history H. For any transaction T and non-transactional access n:*

- *if $T \xrightarrow{\mathsf{DEP}}{}^{*} n$, then there are T' and n' such that $T \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}}{}^{*} T' \xrightarrow{\mathsf{PO}} n' \xrightarrow{\mathsf{CL}}{}^{*} n$;*
- *if $n \xrightarrow{\mathsf{DEP}}{}^{*} T$, then there are T' and n' such that $n \xrightarrow{\mathsf{CL}}{}^{*} n' \xrightarrow{\mathsf{PO}} T' \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}}{}^{*} T$.*

Our method for proving the privatization-safe opacity of a TM (which we illustrate in §7) uses Lemmas 15 and 16 to reduce proving the acyclicity of an opacity graph to proving the absence of cycles in the projection of the graph to transactions, enriched with real-time order edges. The simplified acyclicity check is exactly the one required in the graph characterization of the ordinary opacity [21], allowing us to reuse existing proofs.

In the following we prove Lemmas 15 and 16. We show the existence of the paths required in the lemmas by using CDRF to eliminate $\mathsf{WR}/\mathsf{WW}/\mathsf{RW}$-dependencies between transactions and non-transactional accesses. Each of the dependencies to be eliminated corresponds to a conflict in a matching atomic history, which CDRF guarantees to relate by happens-before. The next lemma exploits this observation.

▶ **Lemma 17.** *Consider an acyclic opacity graph $G = (\mathcal{V}, \mathsf{vis}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{PO}, \mathsf{CL})$ of a consistent CDRF history H. For any transaction T and non-transactional access n:*

1. *if $T \xrightarrow{\mathsf{DEP}} n$, then there are T' and n' such that $T \xrightarrow{\mathsf{DEP}}{}^{*} T' \xrightarrow{\mathsf{PO}} n' \xrightarrow{\mathsf{CL}}{}^{*} n$;*
2. *if $n \xrightarrow{\mathsf{DEP}} T$, then there are T' and n' such that $n \xrightarrow{\mathsf{CL}}{}^{*} n' \xrightarrow{\mathsf{PO}} T' \xrightarrow{\mathsf{DEP}}{}^{*} T$.*

For example, consider an execution of the program in Figure 1 where T_2 reads `false` from `priv` and writes to `x` before n does. The corresponding acyclic graph contains both $T_2 \xrightarrow{\mathsf{WW}_{\mathtt{x}}} n$ and $T_2 \xrightarrow{\mathsf{RW}_{\mathtt{priv}}} T_1 \xrightarrow{\mathsf{PO}} n$. To prove Lemma 17, we lift $<_{\mathsf{po}(H)}$, $<_{\mathsf{ef}(H)}$, $<_{\mathsf{cl}(H)}$ and $<_{\mathsf{hb}(H)}$ from Definition 2 to vertexes of the graph as expected, writing $<_{\mathsf{PO}(H)}$, $<_{\mathsf{EF}(H)}$, $<_{\mathsf{CL}(H)}$ and $<_{\mathsf{HB}(H)}$ for the resulting relations. We also write $\leq$ for their reflexive closure. We rely on the following easy result (proved in [26, §D]).

▶ **Proposition 18.** *In a TDRF history H, for any T and n we have:*

- *if $T <_{\mathsf{HB}(H)} n$, then there are T' and n' such that $T \leq_{\mathsf{EF}(H)} T' <_{\mathsf{PO}(H)} n' \leq_{\mathsf{CL}(H)} n$;*
- *if $n <_{\mathsf{HB}(H)} T$, then there are T' and n' such that $n \leq_{\mathsf{CL}(H)} n' <_{\mathsf{PO}(H)} T' \leq_{\mathsf{EF}(H)} T$.*

Proof of Lemma 17. We only prove part 1, as part 2 can be proven analogously. Assume $T \xrightarrow{\mathsf{DEP}} n$. If $T \xrightarrow{\mathsf{PO}} n$, then $T \xrightarrow{\mathsf{DEP}}{}^{*} T \xrightarrow{\mathsf{PO}} n \xrightarrow{\mathsf{CL}}{}^{*} n$, which trivially concludes the proof. In the following, we consider the remaining case when $\neg(T \xrightarrow{\mathsf{PO}} n)$ and $T \xrightarrow{\mathsf{WR} \cup \mathsf{RW} \cup \mathsf{WW}} n$, so that T and n contain conflicting actions. Let $\mathcal{A}$ denote the following set of pairs (T', n') of a transaction and a non-transactional access:

$$\mathcal{A} \triangleq \{(T', n') \mid \exists L \in \mathsf{lins}(G).\, T \leq_{\mathsf{EF}(L)} T' <_{\mathsf{PO}(L)} n' \leq_{\mathsf{CL}(L)} n\}.$$

By Definition 12, for any $(T', n') \in \mathcal{A}$ we have $T' \xrightarrow{\mathsf{PO}} n' \xrightarrow{\mathsf{CL}}{}^{*} n$. It suffices to show that there is T' such that $T \xrightarrow{\mathsf{DEP}}{}^{*} T'$ and $(T', \_) \in \mathcal{A}$. Proceeding by contradiction, let us assume that this is not the case: for every $(T', \_) \in \mathcal{A}$, there is no edge $T \xrightarrow{\mathsf{DEP}}{}^{*} T'$ in G. Then extending the graph with edges $\{T' \xrightarrow{\mathsf{DEP}} T \mid (T', \_) \in \mathcal{A}\}$ will not introduce a cycle. Hence, there exists a linearization $L \in \mathsf{lins}(G)$ in which every $(T', \_) \in \mathcal{A}$ occurs before T: $\forall T'.\, (T', \_) \in \mathcal{A} \implies T' <_{\mathsf{EF}(L)} T$.

Since, the history H is consistent and has an acyclic opacity graph G, by Lemma 13 we get $L \in \mathsf{lins}(G) \subseteq \mathcal{H}_{\mathsf{atomic}}$. Since H is CDRF, the conflicting pair T and n are ordered by $\mathsf{HB}(L)$. Moreover, since T occurs before n in L and $\mathsf{HB}(L)$ is consistent with the execution order of L, we have $T <_{\mathsf{HB}(L)} n$. From this by Proposition 18, for some T'' and n'' we have $T \leq_{\mathsf{EF}(L)} T'' <_{\mathsf{PO}(L)} n'' \leq_{\mathsf{CL}(L)} n$. Hence, $T \leq_{\mathsf{EF}(L)} T''$ and $(T'', n'') \in \mathcal{A}$. But by the construction of L we have $T'' <_{\mathsf{EF}(L)} T$, which contradicts the definition of ef as a total order on transactions. This contradiction demonstrates the required. ◀

The following result leverages Lemma 17 to show that, for every path between two transactions in an acyclic opacity graph, there is another path replacing some edges involving non-transactional accesses by real-time order edges or transactional dependencies.

▶ **Lemma 19.** *Consider an acyclic opacity graph $G = (\mathcal{V}, \mathsf{vis}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{PO}, \mathsf{CL})$ of a consistent CDRF history H. For any two transactions T and T', if $T \xrightarrow{\mathsf{DEP}}{}^{+} T'$, then there are two transactions T_1 and T_2 such that $T \xrightarrow{\mathsf{DEP}}{}^{*} T_1 \xrightarrow{\mathsf{txDEP} \cup \mathsf{RT}} T_2 \xrightarrow{\mathsf{DEP}}{}^{*} T'$.*

Proof. Assume $T \xrightarrow{\mathsf{DEP}}{}^{+} T'$ and consider the corresponding path in the graph G. If there are no non-transactional accesses on this path, then $T \xrightarrow{\mathsf{txDEP}}{}^{+} T'$, so the lemma holds trivially.

Assume now that there are non-transactional accesses on the path corresponding to $T \xrightarrow{\mathsf{DEP}}{}^{+} T'$. Let n and n' be the first and the last such accesses respectively, and also let T_1' (T_2') be the transaction immediately preceding n (following n') on the path. Since G

is acyclic and CL relates every pair of non-transactional accesses, we must have $n \xrightarrow{\mathsf{CL}}{}^* n'$. Then $T \xrightarrow{\mathsf{DEP}}{}^* T'_1 \xrightarrow{\mathsf{DEP}} n \xrightarrow{\mathsf{CL}}{}^* n' \xrightarrow{\mathsf{DEP}} T'_2 \xrightarrow{\mathsf{DEP}}{}^* T'$. Applying Lemma 17(1) to $T'_1 \xrightarrow{\mathsf{DEP}} n$ and Lemma 17(2) to $n' \xrightarrow{\mathsf{DEP}} T'_2$, we get that there are T_1, n_1, T_2 and n_2 such that:

$$T \xrightarrow{\mathsf{DEP}}{}^* T'_1 \xrightarrow{\mathsf{DEP}}{}^* T_1 \xrightarrow{\mathsf{PO}} n_1 \xrightarrow{\mathsf{CL}}{}^* n \xrightarrow{\mathsf{CL}}{}^* n' \xrightarrow{\mathsf{CL}} n_2 \xrightarrow{\mathsf{PO}} T_2 \xrightarrow{\mathsf{DEP}}{}^* T'_2 \xrightarrow{\mathsf{DEP}}{}^* T'.$$

Then $T \xrightarrow{\mathsf{DEP}}{}^* T_1 \xrightarrow{\mathsf{PO}} n_1 \xrightarrow{\mathsf{CL}}{}^* n_2 \xrightarrow{\mathsf{PO}} T_2 \xrightarrow{\mathsf{DEP}}{}^* T'$. By Definition 12 of PO and CL, T_1 ends before T_2 starts, so that $T_1 \xrightarrow{\mathsf{RT}} T_2$. Then $T \xrightarrow{\mathsf{DEP}}{}^* T_1 \xrightarrow{\mathsf{RT}} T_2 \xrightarrow{\mathsf{DEP}}{}^* T'$, as required. ◀

Proof of Lemma 15. To prove the lemma, we iteratively construct a path in G demonstrating that $T \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}}{}^* T'$. At the k-th iteration we construct a sequence π_k of transactions $T_0, T'_0, T_1, T'_1, \ldots, T_k, T'_k \in \mathcal{V}$ such that:

- $T_0 = T$, $T'_k = T'$, and
- $T_0 \xrightarrow{\mathsf{DEP}}{}^* T'_0 \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}} T_1 \xrightarrow{\mathsf{DEP}}{}^* T'_1 \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}} \ldots \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}} T_k \xrightarrow{\mathsf{DEP}}{}^* T'_k$.

We start the construction with a sequence $\pi_0 = T, T'$, which satisfies the above conditions because $T \xrightarrow{\mathsf{DEP}}{}^* T'$. We stop the construction once we get a sequence π_k such that $T_i = T'_i$ for each $i = 0..k$: in this case the sequence yields a path of the required form. Otherwise, we construct π_{k+1} from π_k as follows. We choose any two transactions T_i and T'_i in π_k such that $T_i \neq T'_i$ and, hence, $T_i \xrightarrow{\mathsf{DEP}}{}^+ T'_i$. By Lemma 19, there are T''_i and T'''_i such that $T_i \xrightarrow{\mathsf{DEP}}{}^* T''_i \xrightarrow{\mathsf{txDEP} \cup \mathsf{RT}} T'''_i \xrightarrow{\mathsf{DEP}}{}^* T'_i$. Then we let $\pi_{k+1} = T_0, T'_0, \ldots, T_i, T''_i, T'''_i, T'_i, \ldots, T_k, T'_k$.

Since G is acyclic, in any π_k the only transactions that can coincide are some consecutive T_i and T'_i. Thus, π_k contains at least $k+1$ distinct transactions. But then our transformation has to stop after at most n steps, where n is the number of transactions in G. ◀

Proof of Lemma 16. We only prove part 1, as part 2 can be proven analogously. Assume $T \xrightarrow{\mathsf{DEP}}{}^* n$. Then there are T'' and n'' such that $T \xrightarrow{\mathsf{DEP}}{}^* T'' \xrightarrow{\mathsf{DEP}} n'' \xrightarrow{\mathsf{DEP}}{}^* n$. By Lemma 17, there are T' and n' such that $T \xrightarrow{\mathsf{DEP}}{}^* T'' \xrightarrow{\mathsf{DEP}}{}^* T' \xrightarrow{\mathsf{PO}} n' \xrightarrow{\mathsf{CL}}{}^* n'' \xrightarrow{\mathsf{CL}}{}^* n$. Then $T \xrightarrow{\mathsf{DEP}}{}^* T' \xrightarrow{\mathsf{PO}} n' \xrightarrow{\mathsf{CL}}{}^* n$. By Lemma 15, $T \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}}{}^* T'$, implying the required. ◀

As we show in [26, §D], the observations in the proofs of the Lemmas 15 and 16 additionally let us establish the following interesting theorem, giving an equivalent formulation of CDRF in terms of dependencies between transactions.

▶ **Theorem 20.** *Given a consistent history H, $\mathsf{CDRF}(H)$ holds if and only if in each acyclic opacity graph $G = (\mathcal{V}, \mathsf{vis}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{PO}, \mathsf{CL}) \in \mathsf{Graph}(H)$ there is a path over edges from $\mathsf{PO} \cup \mathsf{CL} \cup \mathsf{txDEP} \cup \mathsf{RT}(H)$ between every pair of vertexes containing conflicting actions.*

7 Case Study: FencedTL2

In this section we illustrate how Lemmas 15 and 16 enable simple proofs of privatization-safe opacity using an example of a privatization-safe version of TL2 [14]. We give only the key parts of the proof and defer details to [26, §E]. There we also give a proof of privatization-safe opacity of a TM based on two-phase locking [21], which is privatization-safe.

As we noted in §1, the TL2 algorithm by itself is not privatization-safe. The reason is that TL2 executes transactions optimistically, buffering their writes, and flushes them to memory only on commit. Thus, in the example in Figure 1, it is possible for the transaction T_1 to privatize `x` and for n to modify it after T_2 started committing, but before its write to `x` reached the memory, so that T_2's write subsequently overwrites n's write and violates the postcondition. We can make TL2 privatization-safe by modifying its implementation so that it executes a *transactional fence* [31, 35] at the end of every transaction, an implementation we call *FencedTL2*. The fence has a semantics similar to *Read-Copy-Update (RCU)* [32]: it blocks until all the concurrent transactions that were active when the fence was invoked complete, by either committing or aborting. For instance, in the example in Figure 1 executing a transactional fence after T_1 would block the thread until T_2 commits or aborts, thus ensuring that n is not overwritten by T_2's buffered write. The above way of making a TM privatization-safe is used in the GCC compiler [18] (albeit with TinySTM [16] instead of TL2) and has been experimentally evaluated in [36, 37].

To prove privatization-safe opacity of FencedTL2, for every one of its executions we inductively construct an opacity graph (with added real-time order edges) that matches its history. This is done with the help of the following *graph updates*, which specify how and when in the execution to extend the graph:

- At the start of a transaction T, a graph update $\textsc{txinit}(T)$ adds a new vertex T and extends the real-time order with edges $T' \xrightarrow{\mathsf{RT}} T$ for every completed transaction T'.
- At the end of a read operation of a transaction T reading from an object x, a graph update $\textsc{txread}(T, x)$ adds a read dependency $\nu \xrightarrow{\mathsf{WR}_x} T$, where ν is the vertex that wrote the value returned by the read.
- During the commit of a transaction T, TL2 validates the consistency of T's read-set before flushing T's write-set into memory. At the last step of the validation, a graph update $\textsc{txwrite}(T, x)$ adds a write dependency $\nu \xrightarrow{\mathsf{WW}_x} T$ for every object x in the write-set of T, where ν is the vertex that wrote the previous value of x.
- Upon each non-transactional write n to an object x, a graph update $\textsc{ntxwrite}(n, x)$ adds a new vertex n and a write-dependency $\nu \xrightarrow{\mathsf{WW}_x} n$, where ν is the vertex that wrote the previous value of x.
- Upon each non-transactional read n from an object x, a graph update $\textsc{ntxread}(n, x)$ adds a new vertex n and a read dependency $\nu \xrightarrow{\mathsf{WR}_x} n$, where ν is the vertex that wrote the value returned by n.

The updates also add anti-dependencies of the form $\_ \xrightarrow{\mathsf{RW}} T$ induced by new read- and write-dependencies.

At each step of the graph construction we prove that the graph remains acyclic. Then Theorem 14 guarantees that the history of the execution is opaque. We use Lemmas 15 and 16 to reduce the task of proving the graph acyclicity to proving the absence of cycles involving transactions only. To discharge the latter proof obligation, we reuse our previous proof of opacity of TL2 [27], also done via the graph characterization. This proof establishes the following invariant over pairs (H, G) of a history H and a graph G:

- INV_1: H is a consistent history and the relation $\mathsf{txDEP} \cup \mathsf{RT}$ is acyclic.

To enable the reduction from privatization-safe to ordinary opacity, we prove the following invariant, which states the guarantee provided by fences in FencedTL2:

- INV_2: For every uncompleted transaction T and a transaction T', $T \xrightarrow{\mathsf{txDEP}}{}^* T' \xrightarrow{\mathsf{PO}} \_$ does not hold.

An informal justification of INV_2 is as follows. By construction of the graph it is possible to establish that T' can depend on an uncompleted transaction T only when they execute concurrently. In this case, the fence of T' will wait for T to commit or abort, and until then there cannot be any transactions or non-transactional accesses in the thread of T' later in the per-thread order. By Theorem 14, privatization-safe opacity of FencedTL2 follows from

▶ **Theorem 21.** $\forall H \in \text{FencedTL2}.\, \mathsf{CDRF}(H) \implies \exists G.\, (H, G) \in \mathsf{INV}_1 \wedge \mathsf{INV}_2 \wedge \mathsf{acyclic}(G)$.

We prove Theorem 21 by induction on the length of the TM execution inducing H, constructing G as described above and showing that it remains acyclic after each update with the aid of the two invariants. Due to space constraints, we only explain how we prove acyclicity in the case of a graph update TXWRITE, which illustrates the use of Lemmas 15 and 16.

▶ **Lemma 22.** *Let (H', G') be the result of performing an update* TXWRITE(T, x) *on (H, G). Assume that $(H, G), (H', G') \in \mathsf{INV}_1 \wedge \mathsf{INV}_2$ and G is acyclic. Then G' is acyclic too.*

Proof. By contrapositive: we assume that G' contains a simple cycle and show that G' violates either INV_1 or INV_2. The graph update adds an edge of the form $\_ \xrightarrow{\mathsf{WW}_x} T$ and the derived edges of the form $\_ \xrightarrow{\mathsf{RW}_x} T$. Since both kinds of edges end in the same vertex T, they cannot occur in the same simple cycle. Hence, we can consider them separately.

Consider a simple cycle involving a new edge $\nu \xrightarrow{\mathsf{DEP}} T$ for some vertex ν. By our assumption, there must be a reverse path $T \xrightarrow{\mathsf{DEP}}{}^* \nu$ in G. Let us first consider the case when ν is a transaction T'. Since G is acyclic and H is consistent and CDRF, by Lemma 15 the path $T \xrightarrow{\mathsf{DEP}}{}^* T'$ can be reduced to $T \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}}{}^* T'$. Since G' only extends G, the same path is present in G' too. Then $T' \xrightarrow{\mathsf{txDEP}} T \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}}{}^* T'$ is a cycle over transactions in G', which contradicts $(H', G') \in \mathsf{INV}_1$. We now consider the case when ν is a non-transactional access n. Since G is acyclic and H is consistent and CDRF, by Lemma 16 there exist T' and n' such that $T \xrightarrow{\mathsf{txDEP}}{}^* T' \xrightarrow{\mathsf{PO}} n' \xrightarrow{\mathsf{CL}}{}^* n$ holds in G. Note that T is an uncompleted transaction, since it currently performs a graph update. Therefore, $T \xrightarrow{\mathsf{txDEP}}{}^* T' \xrightarrow{\mathsf{PO}} n' \xrightarrow{\mathsf{CL}}{}^* n$ is a contradiction to $(H, G) \in \mathsf{INV}_2$. ◀

8 The Cost of Privatization-Safety

We now present a result about the inherent cost of privatization-safety, by which we mean guaranteeing strongly atomic semantics to TDRF programs. In addition to TM histories, we consider the prefix-closed set of all TM *executions* $\mathcal{X}$, ranged over by φ. Unlike histories, they include internal TM actions that only occur in transactions and are not a part of the TM interface. One type of an internal action are *write-backs* of the form $(a, t, \mathsf{wb}(x, v))$, where $a \in \mathsf{ActionId}$, $t \in \mathsf{ThreadID}$, $x \in \mathsf{Reg}$, and $v \in \mathbb{Z}$. A write-back denotes a transaction of a thread t writing a value v to a register x. We assume that a TM implementation is represented by a pair $(\mathcal{H}, \mathcal{X})$ of a set of histories and a set of executions producing them.

▶ **Definition 23.** *A TM system $(\mathcal{H}, \mathcal{X})$ is* progressive *when for any $\varphi \in \mathcal{X}$ with at most one uncompleted transaction T, if the last interface action by T in φ is a request α, there exists a sequence of internal TM actions φ' by T and a response α' matching α such that $\varphi\varphi'\alpha' \in \mathcal{X}$.*

▶ **Definition 24.** *A TM system $(\mathcal{H}, \mathcal{X})$ has* invisible reads *when for any $\varphi\varphi' \in \mathcal{X}$ such that φ contains at most one uncompleted transaction T and φ' is a sequence of actions corresponding to another uncompleted transaction T' only conflicting with reads by T, if the last interface action by T' is a request α, there exists a sequence of internal TM actions φ'' by T' and a response $\alpha' \neq (\_, \_, \mathsf{aborted})$ matching α such that $\varphi\varphi'\varphi''\alpha' \in \mathcal{X}$.*

Our progressiveness property is analogous to obstruction-freedom [22], requiring a transaction to complete when running solo. Our invisible reads property can be ensured when the TM only writes to thread-local memory upon reading [21]. The FencedTL2 TM from §7 is privatization-safe and has invisible reads, but is not progressive due to its use of fences. As the following theorem shows, this is not accidental.

▶ **Theorem 25.** *A TM system that guarantees strongly atomic semantics to TDRF programs cannot both be progressive and have invisible reads.*

We rely on the following proposition, proved in [26, §G].

▶ **Proposition 26.** *Consider a TM system that guarantees strongly atomic semantics to TDRF programs. If φ is a TM execution of a single atomic block where the latter commits, and $(\_, \_, \mathsf{write}(x, v))$ is its last write request to x, then φ also contains a write-back $(\_, \_, \mathsf{wb}(x, v))$, and all write-backs to x occur in φ after the first write request to x.*

Proof of Theorem 25. The proof is by contradiction. Assume there exists a progressive TM $(\mathcal{H}, \mathcal{X})$ with invisible reads that guarantees strong atomicity to every TDRF program P, so that $[\![P]\!](\mathcal{H}) \preceq [\![P]\!](\mathcal{H}_{\mathsf{atomic}})$. We choose a particular TDRF program P and construct a counterexample trace from $[\![P]\!](\mathcal{H})$ that does not have a matching trace in $[\![P]\!](\mathcal{H}_{\mathsf{atomic}})$. Namely, we consider the following program P, similar to the one in Figure 1:

$$\{\,\texttt{priv} = \texttt{false} \land \texttt{x} = 0\,\}$$

```
l1 = atomic {                  ||  atomic {
  priv = true; } // T1         ||   if (!priv)
if (l1 == committed)           ||     x = 42;
  l2 = x; // n                 ||  } // T2
```

We first consider a single-threaded program executing the atomic block in the right-hand-side thread t_2 of P. The TM always allows the program to execute requests (§2), and the invisible reads property ensures that the TM responds to them without aborting. Therefore, there is an execution $\varphi_2^0 \in \mathcal{X}$ consisting only of actions of the atomic block of t_2 in P ending with a commit-response. By Proposition 26, the execution of φ_2^0 contains a write-back $(\_, t_2, \mathsf{wb}(\texttt{x}, 42))$. Let φ_2 be the prefix of φ_2^0 until the first write-back $w = (\_, t_2, \mathsf{wb}(\texttt{x}, 42))$. By Proposition 26, φ_2 contains a write request to $\texttt{x}$ and, therefore, a preceding response $(\_, t_2, \mathsf{ret}(\texttt{false}))$ to a read from $\texttt{priv}$. The set of TM executions is prefix-closed, so $\varphi_2 w \in \mathcal{X}$.

Note that φ_2 corresponds to a (partial) trace of P. We now let P continue φ_2 by executing the atomic block of the left-hand-side thread t_1. The TM always allows t_1 to execute requests (§2), and the invisible reads property ensures that the TM responds to them without aborting, as they only conflict with t_2's read from $\texttt{priv}$ in φ_2. We thus obtain a sequence of actions φ_1 corresponding to a committed transaction T_1 such that $\varphi_2\varphi_1 \in \mathcal{X}$. We can then execute $n = (\_, t_1, \mathsf{read}(\texttt{x}))(\_, t_1, \mathsf{ret}(0))$, which returns the initial value of $\texttt{x}$ as there has not been any write-back to $\texttt{x}$ yet. We thereby obtain an execution $\varphi_2\varphi_1 n \in \mathcal{X}$ in which thread t_1 of P has executed to completion.

We now let P resume executing the atomic block of thread t_2. Since the TM is progressive, the execution $\varphi_2\varphi_1 n$ can be extended to an execution $\varphi = \varphi_2\varphi_1 n \varphi_2' \in \mathcal{X}$ where the atomic block is completed, yielding a transaction T_2. We first consider the case when T_2 commits in φ. The execution φ corresponds to a trace $\tau \in [\![P]\!](\mathcal{H})$. Since $[\![P]\!](\mathcal{H}) \preceq [\![P]\!](\mathcal{H}_{\mathsf{atomic}})$, there exists a trace $\tau' \in [\![P]\!](\mathcal{H}_{\mathsf{atomic}})$ matching τ. Above we established that φ_2 reads $\texttt{false}$ from $\texttt{priv}$ and, hence, so does T_2. To justify reading this value in τ', T_2 must commit in this trace before T_1 starts and, therefore, before n starts too. Hence, n must observe T_2's write to $\texttt{x}$ in τ', even though it observes the initial value in τ. Then τ' cannot match τ, and this contradiction concludes the proof.

We now consider the case when T_2 aborts in φ. Above we established that $\varphi_2^0 = \varphi_2 w \_ \in \mathcal{X}$, so that $\varphi_2 w \in \mathcal{X}$. Since the TM executes write-backs as atomic writes, if a transaction is interrupted when a write-back w is pending, it proceeds with w once its execution resumes. Hence, it must be the case that φ_2' takes the form of $w\varphi_2''$, so that $\varphi = \varphi_2\varphi_1 n w \varphi_2''$. Since the TM does not impose restrictions on the placement of the non-transactional accesses (§2), it must also allow an execution $\varphi_2 \varphi_1 w n' \varphi_2'' \in \mathcal{X}$, where $n' = (\_, t_1, \mathsf{read}(\mathtt{x}))(\_, t_1, \mathsf{ret}(42))$ returns the value written by w. This execution corresponds to a trace $\tau \in [\![P]\!](\mathcal{H})$. Since $[\![P]\!](\mathcal{H}) \preceq [\![P]\!](\mathcal{H}_{\mathsf{atomic}})$, there exists a trace $\tau' \in [\![P]\!](\mathcal{H}_{\mathsf{atomic}})$ matching τ. In this trace n' reads 42 written by an aborted transaction T_2, which cannot happen under $\mathcal{H}_{\mathsf{atomic}}$. Hence, $\tau' \notin [\![P]\!](\mathcal{H}_{\mathsf{atomic}})$, and this contradiction concludes the proof. ◀

9 Related Work and Discussion

We have previously proposed a notion of DRF for privatization-unsafe TMs and a corresponding variant of opacity that ensure the Fundamental Property [27]. This work considered a more low-level programming model, which required inserting fences after some of the transactions for a program to be DRF. The resulting DRF notion was thus more involved than TDRF. Showing that the simpler TDRF is enough for privatization-safe TMs required us to address new technical challenges, such as the need to generalize TDRF to concurrent histories (to formulate privatization-safe opacity, §4) and to prove the delicate path reduction lemmas linking TDRF with properties of opacity graphs (§6). Furthermore, unlike [27], our results are also applicable to TMs that achieve privatization-safety by means other than fences, such as a lock-based TM we handle in [26, §F]. Our results also suggest a strengthening of those in [27]; we defer the details to [26, §H].

The notion of TDRF we use is a variant of the one proposed by Dalessandro et al. [11]. They also suggested that the notion should satisfy the Fundamental Property, but with strict serializability as the required condition on the TM. As we argued in §4, this condition is too strong, as it does not allow the proofs of TM correctness to benefit from the DRF of programs using it. In this paper we justify the appropriateness of TDRF by proposing a matching TM correctness condition that enables proofs of common TMs and proving the Fundamental Property for it. This also requires us to generalize TDRF to concurrent histories.

In this paper we assumed sequential consistency as a baseline non-transactional memory model. However, transactions are being integrated into languages, such as C++, that have weaker memory models [1]. Transactional sequential consistency, which we use as our strongly atomic semantics, is equivalent to that prescribed by the C++ memory model without relaxed transactions or non-SC atomics [9], and our definition of a data race is given in the axiomatic style used in the C++ memory model [2]. Hence, we believe that in the future our results can be generalized to the wider C++ model, in particular, by weakening the client order in Definition 2 to account for non-SC non-transactional accesses.

Abadi et al. also proposed disciplines for privatization with a formal justification of their safety [3, 4]. However, these disciplines are more restrictive than ours: they either prohibit mixing transactional and non-transactional accesses to the same register [4] or require explicit commands to privatize and publish an object [3]. Such disciplines are particular ways of achieving the more general notion of TDRF that we adopted.

Attiya and Hillel [7] investigated the cost of privatization in progressive TMs. Unlike us, they considered support for privatization to be part of TM interface and did not rely on a formal notion of privatization-safety. They proved the impossibility of supporting privatization in eager TMs, and a lower bound on its implementation cost in lazy TMs. Our Theorem 25 unifies and strengthens their results, as it states the impossibility of providing privatization-safety for all progressive TMs with invisible reads. We also make the results more rigorous by linking them to a formal notion of privatization-safety based on TDRF.

References

1 ISO/IEC. Technical Specification for C++ Extensions for Transactional Memory, 19841:2015, 2015.

2 ISO/IEC. Programming Languages – C++, 14882:2017, 2017.

3 Martín Abadi, Andrew Birrell, Tim Harris, Johnson Hsieh, and Michael Isard. Implementation and Use of Transactional Memory with Dynamic Separation. In *International Conference on Compiler Construction (CC)*, pages 63–77, 2009.

4 Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. *ACM Trans. Program. Lang. Syst.*, 33:2:1–2:50, 2011.

5 Martín Abadi, Tim Harris, and Katherine F. Moore. A Model of Dynamic Separation for Transactional Memory. In *International Conference on Concurrency Theory (CONCUR)*, pages 6–20, 2008. doi:10.1007/978-3-540-85361-9_5.

6 Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. A programming language perspective on transactional memory consistency. In *Symposium on Principles of Distributed Computing (PODC)*, pages 309–318, 2013. doi:10.1145/2484239.2484267.

7 Hagit Attiya and Eshcar Hillel. The Cost of Privatization in Software Transactional Memory. *IEEE Transactions on Computers*, 62(12):2531–2543, 2013.

8 Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *Computer Architecture Letters*, 5(2), 2006.

9 Nathan Chong, Tyler Sorensen, and John Wickerson. The semantics of transactions and weak memory in x86, Power, ARM, and C++. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 211–225, 2018.

10 Luke Dalessandro and Michael L. Scott. Strong Isolation is a Weak Idea. In *Workshop on Transactional Computing (TRANSACT)*, 2009.

11 Luke Dalessandro, Michael L. Scott, and Michael F. Spear. Transactions as the Foundation of a Memory Consistency Model. In *International Symposium on Distributed Computing (DISC)*, pages 20–34, 2010.

12 Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: streamlining STM by abolishing ownership records. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 67–78, 2010. doi:10.1145/1693453.1693464.

13 Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid Transactional Memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 336–346, 2006.

14 David Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *International Symposium on Distributed Computing (DISC)*, pages 194–208, 2006. doi:10.1007/11864219_14.

15 David Dice and Nir Shavit. TLRW: return of the read-write lock. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 284–293, 2010. doi:10.1145/1810479.1810531.

16 P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-Based Software Transactional Memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.

17 Ivana Filipovic, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51-52):4379–4398, 2010.

18 Free Software Foundation. Transactional Memory in GCC. URL: http://gcc.gnu.org/wiki/TransactionalMemory.

19 Rachid Guerraoui, Thomas A. Henzinger, Michal Kapalka, and Vasu Singh. Transactions in the jungle. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 263–272, 2010.

20 Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 175–184, 2008.

21 Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010. doi:10.2200/S00253ED1V01Y201009DCT004.

22 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 522–529, 2003.

23 Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *International Symposium on Computer Architecture (ISCA)*, pages 289–300, 1993. doi:10.1145/165123.165164.

24 Intel Corporation. Intel architecture instruction set extensions programming reference. Chapter 8: Intel transactional synchronization extensions, 2012.

25 Gokcen Kestor, Osman S. Unsal, Adrián Cristal, and Serdar Tasiran. T-Rex: a dynamic race detection tool for C/C++ transactional memory applications. In *European Systems Conference (EuroSys)*, pages 20:1–20:12, 2014.

26 Artem Khyzha, Hagit Attiya, and Alexey Gotsman. Privatization-Safe Transactional Memories (Extended Version). *arXiv CoRR*, 1908.03179, 2019. arXiv:1908.03179.

27 Artem Khyzha, Hagit Attiya, Alexey Gotsman, and Noam Rinetzky. Safe privatization in transactional memory. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages pages 233–245, 2018. Extended version available at arXiv:1801.04249.

28 Sanjeev Kumar, Michael Chu, Christopher J Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 209–220, 2006.

29 H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):8:1–8:14, 2015.

30 Mohsen Lesani, Victor Luchangco, and Mark Moir. Specifying Transactional Memories with Nontransactional Operations. In *Workshop on the Theory of Transactional Memory (WTTM)*, 2013.

31 Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Scalable Techniques for Transparent Privatization in Software Transactional Memory. In *International Conference on Parallel Processing (ICPP)*, pages 67–74, 2008.

32 Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

33 Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *Symposium on Principles of Programming Languages (POPL)*, pages 51–62, 2008. doi:10.1145/1328438.1328448.

34 Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

35 Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *Symposium on Principles of Distributed Computing (PODC)*, pages 338–339, 2007. Extended version appears as Technical Report 915, Computer Science Department, University of Rochester.

36 Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 265–274, 2008. doi:10.1145/1378533.1378582.

37 Tingzhe Zhou, Pantea Zardoshti, and Michael F. Spear. Practical Experience with Transactional Lock Elision. In *International Conference on Parallel Processing (ICPP)*, pages 81–90, 2017.