Reducing Opacity to Linearizability: A Sound and Complete Method

Alasdair Armstrong Brijesh Dongol Simon Doherty
Brunel University London, UK University of Sheffield, UK

July 5, 2021

Abstract

Transactional memory is a mechanism that manages thread synchronisation on behalf of a programmer so that blocks of code execute with an illusion of atomicity. The main safety criterion for transactional memory is opacity, which defines conditions for serialising concurrent transactions.

Proving opacity is complicated because it allows concurrent transactions to observe distinct memory states, while TM implementations are typically based on one single shared store. This paper presents a sound and complete method, based on coarse-grained abstraction, for reducing proofs of opacity to the relatively simpler correctness condition: linearizability. We use our methods to verify TML and NORec from the literature and show our techniques extend to relaxed memory models by showing that both are opaque under TSO without requiring additional fences. Our methods also elucidate TM designs at higher level of abstraction; as an application, we develop a variation of NORec with fast-path reads transactions. All our proofs have been mechanised, either in the Isabelle theorem prover or the PAT model checker.

1 Introduction

Transactional Memory (TM) provides programmers with an easy-to-use synchronisation mechanism for concurrent access to shared data. The basic mechanism is a programming construct that allows one to specify blocks of code as *transactions*, with properties akin to database transactions (atomicity, consistency and isolation) [23]. Like database transactions, a software transaction might encounter interference and abort. Transactions must be invisible to all other transactions until they successfully commit.

Over the last few years, there has been an explosion of research on TM, leading to TM libraries implemented in many programming language libraries (Java, Clojure, C++11), compiler support for TM (G++ 4.7) and hardware support (Intel's Haswell processor). This widespread adoption together with their underlying complexity means that formal verification of TM is an important problem.

The main safety condition for TM is *opacity* [21, 22], which provides conditions for serialising (concurrent) transactions into a sequential order and describes the meaning of this sequential order. Over the years, several methods for verifying opacity have been developed [26, 29, 27, 19, 18, 2, 12]. A difficulty in opacity verification is that it must deal with sequences of memory snapshots that

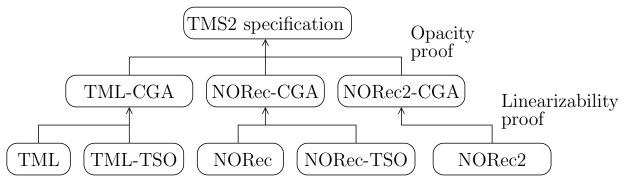

Figure 1: Overview of proofs

reflect the history of all committed transactions. For example, a read-only transaction is allowed to serialise against an earlier snapshot that is different from the current memory.

This paper develops a method for simplifying proofs of opacity by reducing it to the well-known correctness condition *linearizability* [24]. Unlike opacity, linearizability proofs need only concern themselves with a single "current" value of the abstract object. This method is *sound* (any verification using linearizability implies opacity of the original algorithm) and we show that it is also *complete* (for any opaque implementation, it is possible to prove opacity by proving linearizability with respect to an appropriate abstract object). In addition to reducing the complexity of the problem, our approach makes it possible to leverage the rich literature on linearizability verification [16] to verify opacity.

Our method involves development of a coarse-grained abstraction of the TM implementation at hand, where the fine-grained operations of an implementation are abstracted by coarse-grained specification with atomic operations. The first step is to show that this coarse-grained specification is opaque, and the second to show that the implementation does indeed linearize to the coarse-grained abstraction. We then leverage a soundness result to conclude that the original implementation is opaque.

We use a Transactional Mutex Lock (TML) [10] as a running example to introduce the different concepts, then apply our method to this algorithm, i.e., verify it correct by proving linearizability against a coarse-grained abstraction. Then to show the applicability of our approach, we verify the more complex NORec algorithm [11]. In addition, we show that our method scales to relaxed-memory models; we prove opacity of both TML and NORec under TSO memory via a proof of linearizability.

Proof via coarse-grained abstraction has advantages, e.g., it can be used to (more easily) elucidate underlying design principles behind each algorithm. In particular, we prove distinctness between TML and NORec by generating counter-example histories at the level of their coarse-grained abstractions. Such high-level abstractions of each algorithm elucidate design alternatives. In particular, we develop NORec2, a variation of NORec, with read-only transactions that performs fast-path validation when reading previously loaded values. We show that this variation is opaque, and that its design is distinct from the original NORec algorithm (at the coarse-grained level). We develop a fine-grained version of this variation and show that it is a linearizable with respect to this coarse-grained abstraction. We implement this new algorithm in C and evaluate its performance using the STAMP benchmarking suite [7]; our experimental results are discussed in Section 7.

An overview of the proofs we have performed is given in Figure 1, where TML-CGA is the coarse-grained abstraction corresponding to TML. Each proof of opacity has been mechanised in the interactive theorem prover Isabelle [34]. A benefit of these proofs is that they have been proved correct for models of arbitrary size. The linearizability proofs could also have been fully verified in Isabelle. However, we chose to exploit one of the many methods for model checking

Listing 1 The Transactional Mutex Lock (TML) algorithm

```
procedure INIT
    $glb \leftarrow 0$
procedure TXBegin$_t$
    do $loc_t \leftarrow glb$
    until $even(loc_t)$
procedure TXCommit$_t$
    if $odd(loc_t)$ then
        $glb \leftarrow loc_t + 1$
procedure TXRead$_t(a)$
    $v_t \leftarrow mem(a)$
    if $glb = loc_t$ then
        return $v_t$
    else abort
procedure TXWrite$_t(a, v)$
    if $even(loc_t)$ then
        if $!cas(\&glb, loc_t, loc_t{+}1)$
        then abort
        else $loc_t$++
    $mem(a) \leftarrow v$
```

linearizability from the literature. In particular, we use the PAT model checker [37], which provides a process-algebraic encoding of the systems under consideration and allows refinement to be checked automatically. A benefit of the proofs in PAT is that no invariants or simulation relations need to be defined. The method is also able to cope with TSO-encodings of both the TML and NORec algorithms. A surprising result in these proofs is that each implementation in Figure 1 is *equivalent* to its coarse-grained counterpart, i.e., for example we can show that both TML refines TML-CGA *and* TML-CGA refines TML.

This paper is structured as follows. Background material including the TML algorithm and a formal definition of opacity is given in Section 2. Our method of reducing opacity to linearizability, including the soundness and completeness proofs are given in Section 3, which we apply to our TML algorithm in Section 4 (including proofs of opacity and linearizability). The NORec algorithm and its proof is given in Section 5. We give extensions to include relaxed memory in Section 6 and in Section 7, we develop NORec2, demonstrating how reduction via coarse-grained abstraction contributes to TM design.

2 Opacity

This section formalises *opacity* as defined by Guerraoui and Kapalka [22]. Our formalisation mainly follows Attiya *et al.* [4], but we explicitly include the prefix-closure constraint to ensure consistency with other accepted definitions [28, 22, 23]. We introduce TM using the TML algorithm in Section 2.1, formalise histories in Section 2.2 and opacity in Section 2.3.

2.1 Example: Transactional Mutex Lock

To support the concept of transactions, TM usually provide a number of operations to programmers: operations to start (`TXBegin`) or to end a transaction (`TXCommit`), and operations to read or write shared data (`TXRead`, `TXWrite`)[1]. These operations can be called (invoked) from within a program (possibly with some arguments, e.g., the variable to be read) and then will return with a response. Except for operations that start transactions, all other operations might potentially respond with `TXAbort`, thereby aborting the whole transaction.

[1]In general, arbitrary operations can be used here; for simplicity we use reads and writes to variables.

invocations	possible matching responses
$\texttt{TXBegin}_p$	$\overline{\texttt{TXBegin}}_p$
$\texttt{TXCommit}_p$	$\overline{\texttt{TXCommit}}_p$, $\overline{\texttt{TXAbort}}_p$
$\texttt{TXRead}_p(a)$	$\overline{\texttt{TXRead}}_p(v)$, $\overline{\texttt{TXAbort}}_p$
$\texttt{TXWrite}_p(a, v)$	$\overline{\texttt{TXWrite}}_p$, $\overline{\texttt{TXAbort}}_p$

Assume p is a transaction identifier from a set of transactions T, a is an address, and v a value in the address.

Table 1: Events of a TM implementation

We will use the Transactional Mutex Lock (TML) by Dalessandro *et al.* [10] in Listing 1 as a running example. It provides the four types of operations, but operation `TXCommit` in this algorithm will never respond with abort. TML adopts a very strict policy for synchronisation among transactions: as soon as one transaction has successfully written to a variable, all other transactions running concurrently will be aborted when they execute a `TXRead` or `TXWrite` operation. For synchronisation, TML uses a global counter *glb* (initially 0), and each transaction t uses a local variable loc_t to store a local copy of *glb*. Variable *glb* records whether there is a *live writing transaction*, i.e., a transaction which has started, has neither committed nor aborted, and has executed (or is executing) a write operation. More precisely, *glb* is odd if there is a live writing transaction, and even otherwise. Initially, we have no live writing transaction and thus *glb* is 0 (and hence even).

Operation $\texttt{TXBegin}_t$ copies the value of *glb* into its local variable loc_t and checks whether *glb* is even. If so, the transaction is started, and otherwise, the transaction attempts to start again by rereading *glb*. Operation $\texttt{TXRead}_t$ succeeds as long as *glb* equals loc_t (meaning no writes have occurred since the transaction t began), otherwise it aborts the current transaction. The first execution of $\texttt{TXWrite}_t$ attempts to increment *glb* using a *cas* (compare-and-swap), which atomically compares the first and second parameters, and sets the first parameter to the third if the comparison succeeds. If the *cas* attempt fails, a write by another transaction must have occurred, and hence, the current transaction aborts. Otherwise loc_t is incremented (making its value odd) and the write is performed. Note that because loc_t becomes odd after the first successful write, all successive writes that are part of the same transaction will perform the write directly after testing loc_t at line 1. Further note that if the *cas* succeeds, *glb* becomes odd, which prevents other transactions from starting, and causes all concurrent live transactions still wanting to read or write to abort. Thus a writing transaction that successfully updates *glb* effectively locks shared memory. Operation $\texttt{TXCommit}_t$ checks to see if t has written to memory by testing whether loc_t is odd. If the test succeeds, *glb* is set to $loc_t + 1$. At line 8, loc_t is guaranteed to be equal to *glb*, and therefore this update has the effect of incrementing *glb* to an even value, allowing other transactions to begin.

2.2 Histories

As standard in the literature, opacity is defined on the *histories* of an implementation. Histories are sequences of *events* that record all interactions between the TM implementation and its clients. Thus each event is either an invocation or a response of a TM operation. For the implementations we consider in this paper, possible invocation and matching response events are given in Table 1.

We introduce notation $\textbf{TXBegin}_p$ to denote the two-element sequence $\langle \texttt{TXBegin}_p, \overline{\texttt{TXBegin}_p} \rangle$. Similarly, $\textbf{TXWrite}_p(x,v)$ denotes the sequence $\langle \texttt{TXWrite}_p(x,v), \overline{\texttt{TXWrite}_p} \rangle$ and $\textbf{TXRead}_p(x,v)$ denotes $\langle \texttt{TXRead}_p(x), \overline{\texttt{TXRead}_p}(v) \rangle$. We use notation '$\cdot$' for sequence concatenation.

Example 1. *The following history is a possible execution of the TML, where the address x (initially 0) is accessed by two transactions 2 and 3 running concurrently.*

$$\begin{array}{l}\langle \texttt{TXBegin}_3, \texttt{TXBegin}_2, \overline{\texttt{TXBegin}_3}, \overline{\texttt{TXBegin}_2}, \texttt{TXWrite}_3(x,4) \rangle \cdot \\ \textbf{TXRead}_2(x,0) \cdot \langle \overline{\texttt{TXWrite}_3} \rangle \cdot \textbf{TXCommit}_3\end{array}$$

We use the following notation on histories: for a history h, $h|p$ is the projection onto the events of transaction p and $h[i..j]$ the sub-sequence of h from $h(i)$ to $h(j)$ inclusive. We are interested in three different types of histories. At the concrete level the TML implementation produces histories of interleaved events (e.g., h in Example 1). At an intermediate level, we are interested in *alternating histories*, where operations appear atomic, but interleaving may occur at the level of transactions. At the abstract level, we are interested in *sequential histories*, where there is no interleaving at any level—transactions are atomic: completed transactions end before the next transaction starts.

Let ε denote the empty sequence. A history h is *alternating* if $h = \varepsilon$ or h is an alternating sequence of invocation and matching response events starting with an invocation and possibly ending with an invocation. We will assume that $h|p$ is alternating for any history h and transaction p. Note that this does not necessarily mean h is alternating itself. Opacity is defined for well-formed histories, which formalise the allowable interaction between a TM implementation and its clients. A projection $h|p$ of a history h onto a transaction p is *well-formed* iff it is ε or a sequence $s_0 \dots s_m$ where $s_0 = \texttt{TXBegin}_p$ and for all $0 < i < m$, $s_i \notin \{\overline{\texttt{TXBegin}_p}, \overline{\texttt{TXCommit}_p}, \overline{\texttt{TXAbort}_p}\}$. Furthermore, $h|p$ is *committed* whenever $s_m = \overline{\texttt{TXCommit}_p}$ and *aborted* whenever $s_m = \overline{\texttt{TXAbort}_p}$. In these cases, the transaction $h|p$ is *finished*, otherwise it is *live*. A history is *well-formed* iff $h|p$ is well-formed for every transaction p.

Example 2. *The history in Example 1 is well-formed, and contains a committed transaction 3 and a live transaction 2.* □

2.3 Opacity

The basic principle behind the definition of opacity (and similar definitions) is the comparison of a given concurrent history against a sequential one. Within the concurrent history in question, we distinguish between *live*, *committed* and *aborted* transactions. Opacity imposes a number of constraints, that can be categorised into three main types: (1) *ordering constraints*, which describe how events occurring in a concurrent history may be sequentialised; (2) *semantic constraints* that describe validity of a sequential history *hs*; and (3) a *prefix-closure constraint*, which requires that each prefix of a concurrent history can be sequentialised so that the ordering and semantic constraints above are satisfied.

To help formalise these opacity constraints we introduce the following notation. We say a history h is *equivalent* to a history h', denoted $h \equiv h'$, iff $h|p = h'|p$ for all transactions $p \in T$. Further, the *real-time order* on transactions p and q in a history h is defined as $p \prec_h q$ if p is a completed transaction and the last event of p in h occurs before the first event of q.

Sequential history semantics. We now formalise the notion of sequentiality for transactions: a sequential history is alternating and does not interleave events of different transactions. We first define non-interleaved histories, noting that the definition must also cover live transactions.

Definition 3 (Non-interleaved history)**.** *A well-formed history h is* non-interleaved *if transactions do not overlap, i.e., for any transactions p and q and histories h_1, h_2 and h_3, if*

$$h = h_1 \cdot \langle \texttt{TXBegin}_p \rangle \cdot h_2 \cdot \langle \texttt{TXBegin}_q \rangle \cdot h_3$$

and h_2 contains no TXBegin *invocation events, then either h_2 contains a response event e such that $e \in \{\overline{\texttt{TXAbort}}_p, \overline{\texttt{TXCommit}}_p\}$, or h_3 contains no events for a transaction p.*

In addition to being non-interleaved, a sequential history has to ensure that the behaviour is meaningful with respect to the reads and writes of the transactions. For this, we look at each address in isolation and define the notion of a valid sequential behaviour on a single address. To this end, we model shared memory by a set A of addresses mapped to values denoted by a set V. Hence the type $A \to V$ describes the possible states of the shared memory.

Definition 4 (Valid history)**.** *Let $h = \langle ev_0, \ldots, ev_{2n-1} \rangle$ be a sequence of alternating invocation and response events starting with an invocation and ending with a response.*

We say h is valid *if there exists a sequence of states $\sigma_0, \ldots, \sigma_n$ such that $\sigma_0(a) = 0$ for all addresses a, and for all i such that $0 \leq i < n$ and $p \in T$:*

1. *if $ev_{2i} = \texttt{TXWrite}_p(a, v)$ and $ev_{2i+1} = \overline{\texttt{TXWrite}}_p$ then $\sigma_{i+1} = \sigma_i[a := v]$; and*
2. *if $ev_{2i} = \texttt{TXRead}_p(a)$ and $ev_{2i+1} = \overline{\texttt{TXRead}}_p(v)$ then both $\sigma_i(a) = v$ and $\sigma_{i+1} = \sigma_i$ hold; and*
3. *for all other pairs of events $\sigma_{i+1} = \sigma_i$.*

The effect of the writes in a transaction must only be visible if a transaction commits. All other writes must not affect memory. However, all reads must be consistent with previously committed writes. Therefore, only some histories of an object reflect ones that could be produced by an STM. We call these the *legal* histories, and they are defined as follows.

Definition 5 (Legal history)**.** *Let hs be a non-interleaved history, i an index of hs, and hs' be the projection of $hs[0..(i-1)]$ onto all events of committed transactions plus the events of the transaction to which $hs(i)$ belongs. We say hs* is legal at *i whenever hs' is valid. We say hs* is legal *iff it is legal at each index i.*

This allows us to define sequentiality for a single history, which we lift to the level of specifications.

Definition 6 (Sequential history)**.** *A well-formed history hs is* sequential *if it is non-interleaved and legal. We denote by $\mathcal{S}$ the set of all possible well-formed sequential histories.*

Opaque histories. A given concrete history may be incomplete, i.e., consist of pending operation calls. As some of these pending calls may have taken effect, pending operation calls may be completed by adding matching responses. Thus we define a function *extend* that adds matching responses to some of the pending invocations to a history h. There may also be incomplete operation calls that have not taken effect; it is safe to remove the pending invocations. We let $[h]$ denote the history h with all pending invocations removed.

Definition 7 (Opaque history)**.** *A history h is* end-to-end opaque *iff for some $he \in extend(h)$, there exists a sequential history $hs \in \mathcal{S}$ such that $[he] \equiv hs$ and $\prec_{[he]} \subseteq \prec_{hs}$. A history h is* opaque *iff each prefix h' of h is end-to-end opaque; a set of histories $\mathcal{H}$ is* opaque *iff each $h \in \mathcal{H}$ is opaque; and an STM implementation is* opaque *iff its set of histories is opaque.*

In Definition 7, conditions $[he] \equiv hs$ and $\prec_{[he]} \subseteq \prec_{hs}$ establish the ordering constraints and the requirement that $hs \in \mathcal{S}$ ensures the memory semantics constraints. Finally, the prefix-closure constraints are ensured because end-to-end opacity is checked for each prefix of $[he]$.

Example 8. *The history in Example 1 is opaque; a corresponding sequential history is*

$$\mathbf{TXBegin}_2 \cdot \mathbf{TXRead}_2(x, 0) \cdot \mathbf{TXBegin}_3 \cdot \\ \mathbf{TXWrite}_3(x, 4) \cdot \mathbf{TXCommit}_3$$

Note that reordering of $\mathtt{TXRead}_2(x, 0)$ *and* $\mathtt{TXBegin}_3$ *is allowed because their corresponding transactions overlap (even though the operations themselves do not).*

3 Reducing opacity to linearizability

In this section, we show that it is possible to reduce a proof of opacity of an implementation to a proof of linearizability against a coarse-grained abstraction (where each TM operation is atomic). First, we recap the definition of linearizability (Section 3.1), then we present soundness of the result, and also establish that the method is complete (Section 3.2).

3.1 Linearizability

As with opacity, the formal definition of linearizability is given in terms of histories (of invocation/response events); for every concurrent history an equivalent alternating (invocations immediately followed by the matching response) history must exist that preserves real-time order of operations. The *real-time order* on operation calls[2] o_1 and o_2 in a history h is defined as $o_1 \lessdot_h o_2$ if the response of o_1 precedes the invocation of o_2 in h.

Linearizability differs from opacity in that it does not deal with transactions; thus transactions may still be interleaved in a matched alternating history. As with opacity, the given concurrent history may be incomplete, and hence, may need to be extended using *extend* and all remaining pending invocations may need to be removed. We say $lin(h, ha)$ holds iff both $[h] \equiv ha$ and $\lessdot_{[h]} \subseteq \lessdot_{ha}$ hold.

Definition 9 (Linearizability)**.** *A history h is* linearized *by alternating history ha iff there exists a history $he \in extend(h)$ such that $lin(he, ha)$. A concurrent object is linearizable with respect to a specification $\mathcal{A}$ (a set of alternating histories) if for each concurrent history h, there is a history $ha \in \mathcal{A}$ that linearizes h.*

3.2 Soundness and completeness

With linearizability formalised, we now present the two main theorems for our proof method. The first establishes soundness, i.e., states that one can prove opacity of an implementation by first linearizing the concurrent operations, then establishing opacity of the linearized history.

Lemma 10 (Soundness per history [12])**.** *Suppose h is a concrete history. For any alternating history ha that linearizes h, if ha is opaque then h is also opaque.*

The following theorem lifts this existing result to sets of histories (of an implementation).

[2]Note: this is different from the real-time order on transactions defined in Section 2.3

Theorem 11 (Soundness)**.** *Suppose $\mathcal{A}$ is a set of alternating opaque histories. Then a set of histories $\mathcal{H}$ is opaque if for each $h \in \mathcal{H}$, there exists a history $ha \in \mathcal{A}$ and an $he \in extend(h)$ such that $lin(he, ha)$.*

Next we establish completeness of our proof method, i.e., we show that if an implementation is opaque, there is some specification S satisfying opacity, and such that the implementation history is linearizable to S.

Lemma 12 (Existence of Linearization)**.** *If h is an opaque history then there exists an alternating history ha such that $lin(h, ha)$ and ha is end-to-end opaque.*

Proof. From the assumption that h is opaque, there exists an extension $he \in extend(h)$ and a sequential history $hs \in \mathcal{S}$ such that $[he] \equiv hs$ and $\prec_{[he]} \subseteq \prec_{hs}$. Our proof proceeds by transposing operations in hs to obtain an alternating history ha such that $lin(he, ha)$. Our transpositions preserve end-to-end opacity, so ha is end-to-end opaque.

We consider pairs of operations o_t and $o_{t'}$ such that $o_t \lessdot_{hs} o_{t'}$, but $o_{t'} \lessdot_{[he]} o_t$, which we call *mis-ordered pairs*. If there are no mis-ordered pairs, then $lin(he, hs)$, and we are done. Let o_t and $o_{t'}$ be the mis-ordered pair such that the distance between o_t and $o_{t'}$ in hs is least among all mis-ordered pairs. Now, hs has the form $\ldots o_t g o_{t'} \ldots$. Note that g does not contain any operations of transaction t, since if there were some operation o of t in g, then because opacity preserves program order and $o_t \lessdot_{hs} o$, we would have $o_t \lessdot_{[he]} o$. Thus $o, o_{t'}$ would form a mis-ordered pair of lower distance, contrary to hypothesis. For a similar reason, g does not contain any operations of t'. Thus, so long as we do not create a new edge in the opacity order $\prec_{hs}$, we can reorder hs to (1) $\ldots g o_{t'} o_t \ldots$ or (2) $\ldots o_{t'} o_t g \ldots$ while preserving opacity. A new edge can be created only by reordering a pair of begin and commit operations so that the commit precedes the begin. If o_t is not a begin operation, then we choose option (1). Otherwise, note that $o_{t'}$ cannot be a commit, because since $o_{t'} \lessdot_{[he]} o_t$, $t' \prec t$, and thus t could not have been serialised before t'. Since $o_{t'}$ is not a commit, we can choose option (2). Finally, we show that the new history has no new mis-ordered pairs. Assume we took option (1). Then if there is some o in g such that $o_t \lessdot_{[he]} o$ we would have $o_{t'} \lessdot_{[he]} o$, and thus $o, o_{t'}$ would form a narrower mis-ordered pair. The argument for case (2) is symmetric. Thus, we can repeat this reordering process and eventually arrive at an end-to-end opaque history ha that has no mis-ordered pairs, and thus $lin(he, ha)$. ☐

Theorem 13 (Completeness)**.** *If $\mathcal{H}$ is a prefix-closed set of opaque histories, then there is some prefix-closed set of opaque alternating histories $\mathcal{H}'$ such that for each $h \in \mathcal{H}$ there is some $h' \in \mathcal{H}'$ such that $lin(h, ha)$.*

Proof. Let $\mathcal{H}' = \{h'.h'$ is opaque and $\exists h \in \mathcal{H}.lin(h, h')\}$. Note that both the set of all opaque histories and the set of linearizable histories of any prefix-closed set are themselves prefix closed. Thus, $\mathcal{H}'$ is prefix closed. Because $\mathcal{H}'$ is prefix closed, and each element is end-to-end opaque, each element of $\mathcal{H}'$ is opaque. For any $h \in \mathcal{H}$, Lemma 12 implies that there is some $ha \in \mathcal{H}'$. ☐

4 Proving opacity via linearizability

This section describes a proof method for reducing opacity to linearizability using our running TML example. In Section 4.1, we present the coarse-grained abstraction for TML, which we will refer to as TML-CGA. There are two distinct proof steps: (1) proving that TML-CGA is opaque; and (2) proving that TML linearizes to TML-CGA. Both steps could be performed using any of the existing methods in the literature. We opt for a fully mechanised simulation-based method for the opacity proof, step (1) (see Section 4.2); and a model-checking approach for step (2) (see Section 4.3). The combination of the two proofs is much simpler than a single proof of opacity of the original algorithm.

4.1 A coarse-grained abstraction

The coarse-grained abstraction that can be used to prove opacity of the TML is given in Listing 2. Like TML in Listing 1, it uses meta-variables loc_t (local to transaction t) and glb (shared by all transactions). Each operation is however, significantly simpler than the TML operations, and performs the entire operation in a single atomic step.

Listing 2 TML-CGA: Coarse-grained abstraction of TML

```
procedure INIT
    $glb \leftarrow 0$

procedure ATXBegin$_t$
    atomic
        await $even(glb)$
        $loc_t \leftarrow glb$

procedure ATXCommit$_t$
    atomic
        if $odd(loc_t)$ then
            $glb$++

procedure ATXRead$_t(a)$
    atomic
        if $glb = loc_t$ then
            return $mem(a)$
        else abort

procedure ATXWrite$_t(a, v)$
    atomic
        if $glb \neq loc_t$ then
            abort
        if $even(loc_t)$ then
            $loc_t$++; $glb$++
            $mem(a) \leftarrow v$
```

4.2 Opacity of the coarse-grained abstraction

Several methods for proving opacity have been developed, and we are free to choose any of these methods. We leverage two existing results from the literature: the TMS2 specification by Doherty *et al.* [15], and the mechanised proof that TMS2 is opaque by Lesani *et al.* [28]. Using these results, it is sufficient that we prove trace refinement (i.e., trace inclusion of visible behaviour) between TML-CGA and the TMS2 specification. The rigorous nature of these existing results, means that a mechanised proof of refinement against TMS2 will also comprise a rigorous proof of opacity of TML-CGA.

TMS2 is formalised using input/output automata [31], and hence, our formalisations will also use IOA. Moreover, Müller [33] has mechanised the IOA theory (including its simulation rules) in Isabelle, which is now part of the standard Isabelle distribution [34]. We thus chose to carry out our proofs within Isabelle. First we define I/O automata.

Definition 14. *An* I/O automaton (IOA) *is a labelled transition system A with a set of states Σ_A, a set of actions $acts(A)$ (partitioned into internal and external actions), a set of start states $start(A) \subseteq \Sigma_A$ and a transition relation $trans(A) \subseteq \Sigma_A \times acts(A) \times \Sigma_A$ (so that the actions label the transitions).*

Next we formalise refinement and a proof method for it based on forward simulation. An *execution* of an IOA A is a sequence σ of alternating states and actions, beginning with a state in $\mathit{start}(A)$, such that for all states σ_i except the last, $(\sigma_i, \sigma_{i+1}, \sigma_{i+2}) \in \mathit{trans}(A)$. A *reachable* state of A is a state appearing in an execution of A. An *invariant* of A is a predicate satisfied by all reachable states of A. A *trace* of A is any sequence of (external) actions obtained by restricting the actions of A to its external actions. The set of traces of A represents A's externally visible behaviour. We say IOA C *refines* A, denoted $A \sqsubseteq C$, iff every trace of C is also a trace of A. We say A is *equivalent* to C iff both $A \sqsubseteq C$ and $C \sqsubseteq A$.

In our setting, each externally visible behaviour consists of the sequence of invoke and response events, including the input/output values of reads and writes. Since TMS2 has been specified to only allow inputs and outputs that result from opaque transactions, a proof that TML-CGA is a refinement of TMS2 also implies opacity of TML-CGA.

The standard way of verifying a refinement is to use a *forward simulation* between the implementation and the specification, as this allows one to verify the refinement in a step-wise manner. We let Σ_A^E and Σ_A^I denote the external and internal actions of IOA A, respectively. Writing $cs \xrightarrow{a}_C cs'$ for $(cs, a, cs') \in \mathit{trans}(C)$, we define the following.

Definition 15. *A* forward simulation *from a concrete IOA C to an abstract IOA A is a relation $R \subseteq \Sigma_C \times \Sigma_A$ such that each of the following holds.*

Initialisation.

$$\forall cs \in \mathit{start}(C) \bullet \exists as \in \mathit{start}(A) \bullet (cs, as) \in R$$

External step correspondence.

$$\begin{array}{l}\forall cs \in \mathit{reach}(C), as \in \mathit{reach}(A), a \in \Sigma_C^E, cs' \in \Sigma_C \bullet \\ \quad (cs, as) \in R \wedge cs \xrightarrow{a}_C cs' \Rightarrow \\ \qquad \exists as' \in \Sigma_A \bullet (cs', as') \in R \wedge as \xrightarrow{a}_A as'\end{array}$$

Internal step correspondence.

$$\begin{array}{l}\forall cs \in \mathit{reach}(C), as \in \mathit{reach}(A), a \in \Sigma_C^I, cs' \in \Sigma_C \bullet \\ \quad (cs, as) \in R \wedge cs \xrightarrow{a}_C cs' \Rightarrow \\ \qquad (cs', as) \in R \vee \\ \qquad \exists as' \in \Sigma_A, a' \in \Sigma_A^I \bullet (cs', as') \in R \wedge as \xrightarrow{a'}_A as'\end{array}$$

The TMS2 specification. The TMS2 specification [15] is designed to capture the structural patterns common to most TM implementations. The actions of TMS2 are given in Figure 2. We use notation '$\oplus$' to denote functional override. For each transition, the first line gives the action name. — names of the form $\mathit{inv}_t(\texttt{Op})$ and $\mathit{resp}_t(\texttt{Op})$ are external invocation and response actions of the operation `Op`, by transaction t respectively; all others names denote internal actions. The transition is *enabled* if all its preconditions, given after the keyword Pre, hold in the current state. State modifications (effects) of a transition are given as assignments after the keyword Eff.

Like opacity, TMS2 guarantees that transactions satisfy two critical requirements: (*R1*) all reads and writes of a transaction work with a *single consistent memory snapshot* that is the result of all previously committed transactions, and (*R2*) the *real-time order* of transactions is preserved.

```
$inv_t(\texttt{TXBegin})$
Pre: $status_t = \text{notStarted}$
Eff: $status_t := \text{beginPending}$
     $beginIdx_t := maxIdx$

$resp_t(\texttt{TXBegin})$
Pre: $status_t = \text{beginPending}$
Eff: $status_t := \text{ready}$

$inv_t(\texttt{TXRead}(a))$
Pre: $status_t = \text{ready}$
Eff: $status_t := \text{doRead}(a)$

$resp_t(\texttt{TXRead}(v))$
Pre: $status_t = \text{readResp}(v)$
Eff: $status_t := \text{ready}$

$inv_t(\texttt{TXWrite}(a, v))$
Pre: $status_t = \text{ready}$
Eff: $status_t := \text{doWrite}(a, v)$

$resp_t(\texttt{TXWrite})$
Pre: $status_t = \text{writeResp}$
Eff: $status_t := \text{ready}$

$inv_t(\texttt{TXCommit})$
Pre: $status_t = \text{ready}$
Eff: $status_t := \text{doCommit}$

$resp_t(\texttt{TXCommit})$
Pre: $status_t = \text{commitResp}$
Eff: $status_t := \text{committed}$

$resp_t(\texttt{TXAbort})$
Pre: $status_t \notin \{\text{notStarted}, \text{ready}, \text{commitResp}, \text{committed}, \text{aborted}\}$
Eff: $status_t := \text{aborted}$

$\texttt{DoCommitRO}_t(n)$
Pre: $status_t = \text{doCommit}$
     $dom(wrSet_t) = \emptyset$
     $validIdx_t(n)$
Eff: $status_t := \text{commitResp}$

$\texttt{DoCommitW}_t$
Pre: $status_t = \text{doCommit}$
     $rdSet_t \subseteq latestMem$
Eff: $status_t := \text{commitResp}$
     $memSeq :=$
       $memSeq \oplus newMem_t$

$\texttt{DoRead}_t(a, n)$
Pre: $status_t = \text{doRead}(a)$
     $a \in dom(wrSet_t) \vee$
       $validIdx_t(n)$
Eff: if $a \in dom(wrSet_t)$ then
       $status_t :=$
         $\text{readResp}(wrSet_t(a))$
     else
       $v := memSeq(n)(a)$
       $status_t := \text{readResp}(v)$
       $rdSet_t :=$
         $rdSet_t \oplus \{a \mapsto v\}$

$\texttt{DoWrite}_t(a, v)$
Pre: $status_t = \text{doWrite}(a, v)$
Eff: $status_t := \text{writeResp}$
     $wrSet_t :=$
       $wrSet_t \oplus \{a \mapsto v\}$

where $maxIdx \mathrel{\widehat{=}} max(dom(memSeq))$
      $latestMem \mathrel{\widehat{=}} memSeq(maxIdx)$
      $newMem_t \mathrel{\widehat{=}} \{maxIdx + 1 \mapsto (latestMem \oplus wrSet_t)\}$
      $validIdx_t(n) \mathrel{\widehat{=}} beginIdx_t \leq n \leq maxIdx \wedge$
                          $rdSet_t \subseteq memSeq(n)$
```

Figure 2: The transitions of TMS2

To ensure (*R1*), the state of TMS2 includes $\langle memSeq(0), \ldots\ memSeq(maxIdx)\rangle$, which is a sequence of all possible memory snapshots. Initially the sequence consists of one element, the initial memory $memSeq(0)$. Committing writer transactions append a new memory *newMem* to this sequence (cf. $\texttt{DoCommitW}_t$), by applying the writes of the transaction to the last element

$\mathit{memSeq}(\mathit{maxIdx})$. To ensure that the writes of a transaction are not visible to other transactions before committing, TMS2 uses a *deferred update* semantics: writes are stored locally in the transaction t's write set wrSet_t and only published to the shared state when the transaction commits. Note that this does not preclude implementations with eager writes (like TML-CGA). However, to ensure opacity, such eager implementations must guarantee that writes are not observable until after the writing transaction has committed.

All reads in TMS2 must be consistent (i.e., occur from a single memory snapshot), therefore each transaction t keeps track of all its reads from memory in a read set rdSet_t. A read of address a by transaction t checks that either a was previously written by t itself (**then** branch of $\texttt{DoRead}_t(a)$), or that all values read so far, including a, are from the same memory snapshot n, where $\mathit{beginIdx}_t \leq n \leq \mathit{maxIdx}$ (predicate $\mathit{validIdx}_t(n)$ from the precondition, which must hold in the **else** branch). In the former case the value of a from wrSet_t is returned, and in the latter the value from $\mathit{memSeq}(n)$ is returned and the read set is updated. The read set of t is also validated when a transaction commits (cf. $\texttt{DoCommitRO}_t$ and $\texttt{DoCommitW}_t$). Note that when committing, a read-only transaction may read from a memory snapshot older than $\mathit{memSeq}(\mathit{maxIdx})$, but a writing transaction must ensure that all reads in its read set are from most recent memory (i.e. $\mathit{latestMem}\ \mathit{memSeq}(\mathit{maxIdx})$), since its writes will update the memory sequence with a new snapshot.

To ensure (*R2*), if a transaction t' commits before transaction t starts, then the memory that t reads from must include the writes of t'. Thus, when starting a transaction (cf. $\mathit{inv}_t(\texttt{TXBegin})$), t saves the current last index of the memory sequence, maxIdx, into a local variable $\mathit{beginIdx}_t$. When t performs a read, the check $\mathit{validIdx}_t(n)$ ensures that that the snapshot $\mathit{memSeq}(n)$ used has $\mathit{beginIdx}_t \leq n$, which implies that the writes of t' are included.

Theorem 16. TML-CGA *is opaque.*

Proof. We prove forward simulation between the IOA representation of TML-CGA and the TMS2 specification. To construct the simulation relation between TML-CGA and TMS2, we start by defining a (partial) *step-correspondence* function sc, mapping the internal actions of TML-CGA to the internal actions of TMS2. We let $\perp$ denote an undefined value.

$$\begin{aligned}
sc(\texttt{ATXRead}_t(a)) &= \text{if } \mathit{loc}_t = \mathit{glb} \text{ then } \texttt{DoRead}_t(a, \lfloor \mathit{loc}_t/2 \rfloor) \\
&\quad\ \text{else } \perp \\
sc(\texttt{ATXWrite}_t(a, v)) &= \text{if } \mathit{loc}_t = \mathit{glb} \text{ then } \texttt{DoWrite}_t(a, v) \\
&\quad\ \text{else } \perp \\
sc(\texttt{ATXCommit}_t) &= \text{if } \mathit{even}(\mathit{loc}_t) \text{ then} \\
&\qquad\quad \texttt{DoCommitRO}_t(\lfloor \mathit{loc}_t/2 \rfloor) \\
&\quad\ \text{else } \texttt{DoCommitW}_t
\end{aligned}$$

In addition, we define a global relation between the concrete (i.e., TML-CGA) and abstract (i.e., TMS2) states. This relation states that the concrete store, *mem*, is equal to the latest store in TMS2, with the all the concrete transaction's writes applied to it. This is necessary, because TML is an eager algorithm where writes are applied immediately, while TMS2 has an write set wherein writes are cached. Furthermore, there is a per-transaction simulation relation which states that for each transaction t, each of the following holds. These relate variables loc_t and glb from TML-CGA to variables of TMS2.

- loc_t is even iff in the write set of TMS2, wrSet_t, is empty.

```
ATXBegin($t$) = $inv$.TXBegin.$t$ → ATXBeginLoop($t$);

ATXBeginLoop($t$) =
  ifa($even(glb)$){
    tau{$loc[t] = glb$;} → $ret$.TXBegin.$t$ → Skip
  } else {
    tau → ATXBeginLoop($t$)
  };
```

Figure 3: TML-CGA ATXBegin$_t$ procedure in PAT

- If loc_t is odd, then in TMS2, t is the currently active writer. Furthermore, glb is equal to loc_t.
- The maximum index of $memSeq$ (the sequence of TMS2 stores) is less than or equal to $\lfloor loc_t/2 \rfloor$.
- Each read of t (from the TML-CGA store mem) is consistent from with the store in $memSeq$ at position $\lfloor loc_t/2 \rfloor$.

The step correspondence function, global relation, and per transaction relation are combined into the overall simulation relation. The proof is fully mechanised in Isabelle [3]. □

4.3 Linearizability against coarse-grained abstraction

Having established opacity of TML-CGA, we can now focus on establishing linearizability between TML and TML-CGA, which by Theorem 11 will ensure opacity of TML. We are free to use any of the available methods from the literature to prove linearizability [16]. We opt for a model-checking approach (as opposed to full verification), which provides assurances of linearizability for finite models. Part of our motivation is to show that model checking indeed becomes a feasible technique for verifying opacity, leveraging one of the many methods that have been developed over the years [30, 8, 6, 39]. It is also possible to use static analysis tools (e.g., [38]) or to perform full verification (e.g., [36]).

We use the PAT model checker [37], which enables one to verify trace refinement (in a manner that guarantees linearizability) without having to explicitly define invariants, refinement relations, or linearization points of the algorithm. Interestingly, the model checker additionally shows that, for the bounds tested, TML is *equivalent* to TML-CGA, i.e., both produce exactly the same set of observable traces (see Lemma 17 below).

PAT allows one to specify algorithms using a CSP-style syntax [25]. However, in contrast to conventional CSP, events in PAT (including τ events) are arbitrary programs assumed to execute atomically—as such they can directly modify shared state, and do not have to communicate via channels with input/output events as in other CSP based model checkers (c.f., FDR3 [17]). This enables our transactional memory algorithms to be implemented very naturally in PAT. As an example, we give the PAT encoding of the `ATXBegin`$_t$ operation from TML-CGA in Figure 3, where inv.`TXBegin`.t and ret.`TXBegin`.t are observable events corresponding to invoking and returning from a begin operation. Internal actions are specified using the `tau` keyword, and `ifa` is a built-in keyword that evaluates the test and executes the next event as a single atomic step. Thus, `ATXBegin`(t) fires an (observable) inv.`TXBegin`.t event then executes as `ATXBeginLoop`(t). If glb is even, it fires the observable ret.`TXBegin`.t event and terminates by executing `Skip`, otherwise it retries `ATXBeginLoop`(t).

Note that interleaving may occur between the external events $inv.\texttt{TXBegin}.t$ and $ret.\texttt{TXBegin}.t$. However, because the main action occurs as a single atomic step, this encoding corresponds to Lynch's *canonical automata* [32], which are guaranteed to be linearizable with respect to the atomic TML-CGA where the invocation, main action and atomic object are executed as a single atomic step. Such canonical encodings are also used by Doherty et al. for verifying concurrent data structures [14] and is the prescribed method of proving linearizability in PAT [30].

The overall behaviour of the algorithm can be specified as the interleaving of N transactions where each transaction begins and then does some number of reads and writes before either committing successfully or aborting.

$$\texttt{Transaction}(t) = \texttt{ATXBegin}(t); \texttt{ATXReadWrites}(t);$$
$$\texttt{TML\_CGA}() = |||t : 0..N-1 @ \texttt{Transaction}(t);$$

Once both the coarse-grained and fine-grained algorithms have been implemented within PAT, trace equivalence (and thus linearizability) can be checked automatically via refinement using:

```
#assert TML() refines TML-CGA();
#assert TML-CGA() refines TML();
```

Obviously, this does not give us a full proof that TML linearizes to TML-CGA, as model-checking of course only checks up to a certain number of transactions with a limited amount of memory.

We thus obtain the following lemma, where constant *SIZE* denotes the size of the memory (i.e., number of addresses) and constant V for the possible values in these addresses. The proof is via model checking using PAT [3].

Lemma 17. *For bounds* $N = 3$, $\mathit{SIZE} = 4$, *and* $V = \{0, 1, 2, 3\}$, *as well as* $N = 4$, $\mathit{SIZE} = 2$, *and* $V = \{0, 1\}$, TML *is equivalent to* TML-CGA.

5 The NORec algorithm

In this section, we show that the method scales to more complex algorithms. In particular, we verify the NORec algorithm by Dalessandro *et al.* [11] (see Listing 3), which is one of the best performing STMs that provides both privatisation and publication safety.

The proof steps for NORec proceeds as with TML. Namely, we construct a coarse-grained abstraction, NORec-CGA (see Listing 4), verify that NORec-CGA is opaque, then show that NORec linearizes to NORec-CGA. As with TML, we do not perform a full verification of linearizability, but rather, model check the linearizability part of the proof using PAT.

The proof that NORec-CGA is opaque proceeds via forward simulation against a variant of TMS2 (TMS3), which does not require read-only transactions to validate during their commit, matching the behaviour of NORec more closely. In particular, TMS3 is identical to TMS2 except that $\texttt{DoCommitRO}_t(n)$ in the TMS2 transition relation is replaced by

$\texttt{DoCommitRO}_t$
Pre: $status_t = \text{doCommit} \wedge dom(wrSet_t) = \emptyset$
Eff: $status_t := \text{commitResp}$

where $validIdx_t(n)$ is no longer required in the precondition. Making this change to TMS2 greatly simplifies the simulation relation for NORec-CGA. We have verified within Isabelle that TMS3 is equivalent to the standard definition of TMS2 from Figure 2.

Listing 3 NORec pseudocode

```
1:  procedure TXBegin$_t$
2:    do $loc_t \leftarrow glb$
3:    until $even(loc_t)$

4:  procedure Validate$_t$
5:    while true do
6:      $time_t \leftarrow glb$
7:      if $odd(time_t)$ then
8:        goto 6
9:      for $a \mapsto v \in rdSet_t$ do
10:       if $mem(a) \neq v$ then
11:         abort
12:     if $time_t = glb$ then
13:       return $time_t$

14: procedure TXCommit$_t$
15:   if $wrSet_t = \emptyset$ then return
16:   while $!cas(glb, loc_t, loc_t + 1)$ do
17:     $loc_t \leftarrow$ Validate$_t$
18:   for $a \mapsto v \in wrSet_t$ do
19:     $mem(a) \leftarrow v$
20:   $glb \leftarrow loc_t + 2$

21: procedure TXWrite$_t(a, v)$
22:   $wrSet_t \leftarrow$
          $wrSet_t \oplus \{a \mapsto v\}$

23: procedure TXRead$_t(a)$
24:   if $a \in dom(wrSet_t)$ then
25:     return $wrSet_t(a)$
26:   $v_t \leftarrow mem(a)$
27:   while $loc_t \neq glb$ do
28:     $loc_t \leftarrow$ Validate$_t$
29:     $v_t \leftarrow mem(a)$
30:     $rdSet_t \leftarrow$
            $rdSet_t \oplus \{a \mapsto v_t\}$
31:     return $v_t$
```

Theorem 18. *TMS3 is equivalent to TMS2.*

Proof. We first prove that TMS3 refines TMS2 (soundness) by forward simulation. The proof is straightforward, as the simulation relation between the two automata is the equality relation, except in the case where we must show that `DoCommitRO`$_t$ in TMS3 simulates `DoCommitRO`$_t(n)$ in TMS2. In this case we must provide a valid index n to show that TMS3 simulates TMS2. In order to accomplish this, we prove an additional invariant to TMS3 which states that there is always some valid index in the stores list for any *in-flight transaction*, i.e., a transaction that has completed its begin operation. Of course, this invariant would also hold for TMS2 itself. Next, we show that TMS2 refines TMS3 (completeness), which also proceeds via forward simulation. Once again, we use equality as the forward simulation relation, and no new invariants need to be introduced. As with our other proofs, the Isabelle code may be downloaded [3]. □

We are now ready to prove opacity of NORec-CGA.

Theorem 19. NORec-CGA *is opaque.*

Proof. As was the case for TML-CGA (see Theorem 16), there is a simple correspondence between the internal actions of NORec-CGA and the internal actions of TMS3 given by a (partial) step

Listing 4 NORec-CGA: Coarse-grained abstraction of NORec

```
procedure ATXBegin$_t$
  return

procedure ATXCommit$(t)$
  atomic
    if $wrSet_t = \emptyset$ then return
    else if $rdSet_t \subseteq mem$ then $mem \leftarrow mem \oplus wrSet_t$
    else abort

procedure ATXWrite$_t(a, v)$
  $wrSet_t \leftarrow wrSet_t \oplus \{a \mapsto v\}$

procedure ATXRead$_t(a)$
  atomic
    if $a \in dom(wrSet_t)$ then return $wrSet_t$
    else if $rdSet_t \subseteq mem$ then return $mem(a)$
    else abort
```

correspondence function:

$$\begin{aligned}
sc(\mathtt{ATXRead}_t(a)) &= \text{if } a \in dom(wrSet_t) \vee rdSet_t \subseteq mem \\
&\quad\ \text{then } \mathtt{DoRead}_t(a, N) \\
&\quad\ \text{else } \bot \\
sc(\mathtt{ATXWrite}_t(a, v)) &= \mathtt{DoWrite}_t(a, v) \\
sc(\mathtt{ATXCommit}_t) &= \text{if } wrSet_t = \emptyset \text{ then } \mathtt{DoCommitRO}_t \\
&\quad\ \text{else } \mathtt{DoCommitW}_t
\end{aligned}$$

where N is the number of previous transactions that have successfully committed. This is implemented as an auxiliary variable in the concrete CGA that is incremented whenever it performs the `ATXCommit` action. The rest of the simulation relation is considerably simpler than for TML-CGA—we only require that the read and write set of the concrete and the abstract states are the same for each transaction, that the number of commits N is equal to the maximum index in the TMS3 stores list and that the latest TMS3 store is equal to the concrete NORec-CGA store. With the simulation relation given by sc and these properties, the simulation proof against TMS3 is relatively straightforward. $\square$

TMS3 and NORec-CGA are quite similar in many respects. They both use read and write sets in the same way, and write-back lazily during the commit. The only additional information needed in the simulation is keeping track of the number of successful commits in NORec-CGA. Thus, the simulation relation and refinement proof is easier than the proof between TML-CGA and TMS2.

Next, we have the following lemma, which is proved via model checking using PAT [3].

Lemma 20. *For bounds $N = 2$, $SIZE = 2$ and $V = \{0, 1\}$,* NORec *is equivalent to* NORec-CGA*.*

Proving opacity directly, i.e., by showing NORec directly implements TMS3 would be much trickier as we would need to concern ourselves with the locking mechanism it employs during the commit to guarantee that the write-back occurs atomically. However, this locking mechanism is effectively only being used to guarantee linearizability of the NORec commit operation, so it need not occur in the opacity proof.

Lesani *et al.* directly verified opacity of NORec [27] via simulation against the TMS2 specification. In comparison to our approach, Lesani *et al.* introduce several layers of intermediate automata, performing the full simulation proof in a step-wise manner. Each of layer adds additional complexity and design elements of the NORec algorithm to the abstract TMS2 specification. A full (in-depth) comparison of against this existing proof [27] has however not been possible because these details are not publicly available. In contrast, we have developed a simple coarse-grained abstraction NORec-CGA that generates opaque traces, reducing the proof of opacity for NORec to linearizability against NORec-CGA.

6 Relaxed memory

We now demonstrate that our method naturally extends to reasoning about opacity of TM implementations under relaxed memory. We will focus on TSO in this Section, but our arguments and methods could be extended to other memory models. Note that we cannot employ a data-race freedom argument [1] to show that TML or NOrec running on TSO are equivalent to sequentially consistent versions of the algorithms. This is because transactional reads can race with the writes of committing transactions (this is true even when we consider the weaker *triangular-race freedom* condition of [35]). This racy behaviour is typical for software transactional memory implementations.

There are two possibilities for verifying our TM algorithms on TSO. (1) Leveraging a proof of opacity of the implementation under sequential consistency then showing that the relaxed memory implementation refines this sequentially consistent implementation. (2) Showing that the implementation under relaxed memory linearizes to the coarse-grained abstraction directly. This approach simply treats an implementation executing under a particular memory model as an alternative implementation of the CGA algorithm in question.

In this paper, we follow the second approach, which shows that model checking linearizability of TSO implementations against a coarse-grained abstraction is indeed feasible. We verify both TML and NORec under TSO within the PAT model checker. To do this we encode a buffer for each transaction where writes to main memory are cached, a separate `Flusher`() process placed in parallel with the transactions non-deterministically flushes buffers at random, as shown:

$$\texttt{Flusher}() = []i : 0..(N-1)@(\texttt{Flush}(i); (\texttt{Skip}[]\texttt{Flusher}()));$$
$$\texttt{TML}() = (|||t : 0..N-1@\texttt{Transaction}(t))|||\texttt{Flusher}();$$

Writes by one transaction will not become visible to another transaction until they are flushed. Due to the significantly increased state-space of the model with these added buffers, checking the relaxed-memory versions of the algorithms takes significantly longer than the ordinary fine-grained algorithms.

Due to the transitivity of trace inclusion, the proof proceeds by showing that the concrete implementation that executes using relaxed memory semantics linearizes to its corresponding coarse-grained abstraction.

We use constant *BUFSIZE* to bound the maximum size of the local buffer for each transaction.

Lemma 21. *For bounds $N = 2$, SIZE $= 2$, BUFSIZE $= 2$ and $V = \{0, 1\}$,* TML *under TSO is equivalent to* TML-CGA *and* NORec *under TSO is equivalent to* NORec-CGA.

Listing 5 NORec2: Coarse-grained `ATXRead` operation and implementation `TXRead`

```
procedure ATXRead_t(a)
  atomic
    if $a \in dom(wrSet_t)$ then return $wrSet_t(a)$
    else if $a \in dom(rdSet_t)$ then return $rdSet_t(a)$
    else if $rdSet_t \subseteq mem$ then return $mem(a)$
    else abort
```

```
procedure TXRead_t(a)
  if $a \in dom(wrSet_t)$ then return $wrSet_t(a)$
  if $a \in dom(rdSet_t)$ then return $rdSet_t(a)$
  $v_t \leftarrow mem(a)$
  while $loc_t \neq glb$ do
    $loc_t \leftarrow$ Validate_t
    $v_t \leftarrow mem(a)$
    $rdSet_t \leftarrow rdSet_t \oplus \{a \mapsto v_t\}$
    return $v_t$
```

7 TM design

Our method of coarse-grained abstraction can potentially contribute to TM development and allows differences in design to be distinguished at a higher level of abstraction. Examining the coarse-grained NORec algorithm, we see that we can avoid validation in the `ATXRead` and `TXRead` operations if we have already previously read the value and stored it in the read set — here we can simply return the value from the read set. This variant in CGA is shown in Listing 5. Under the assumption that validation is a more expensive operation than checking membership of the read set this adds an additional fast path for repeated reads, potentially increasing performance. In this section we show that we were able to quickly verify this variant of the algorithm. Furthermore, we implemented this variant of NORec in the STAMP benchmark suite [7] to compare its performance with the standard NORec algorithm.

Theorem 22. NORec2-CGA *is opaque.*

Proof. The proof for opacity of NORec2-CGA is largely the same as for NORec-CGA. The only significant difference is that we need an additional auxiliary variable at the concrete level to keep track of the number of validating reads that have occurred in the CGA. This is due to how for the ordinary NORec-CGA, the commit of a read-only transaction can be re-ordered to it's last read, while for NORec2-CGA it can be reordered to the last validating read. For details we refer the reader to our Isabelle implementation [3]. □

Lemma 23. *For bounds $N = 2$, $SIZE = 2$ and $V = \{0, 1\}$,* NORec2 *is linearizable with respect to* NORec2-CGA.

We now show that the three coarse-grained abstractions we have considered in this paper (Figure 1) are pairwise distinct.

Lemma 24. TML-CGA*,* NORec-CGA*,* NORec2-CGA *are pairwise distinct.*

Proof. We automatically generate counter-examples using the PAT model checker. First we show TML-CGA is distinct from both NORec-CGA and NORec2-CGA. History h_1 below is allowed by TML-CGA but not allowed by NORec-CGA or NORec2-CGA. TML-CGA only allows one writer at any time, and hence, transaction 1 aborts. However, in NORec-CGA, a write operation never aborts because they are cached in a transaction-local read set.

$$\begin{aligned} h_1 = {} & \mathbf{TXBegin}_0 \cdot \langle \mathtt{TXWrite}_0(x, 0) \rangle \cdot \mathbf{TXBegin}_1 \cdot \\ & \langle \mathtt{TXWrite}_1(x, 0), \mathtt{TXAbort}_1 \rangle \end{aligned}$$

History h_2 below is valid for both NORec-CGA and NORec2-CGA but invalid for TML-CGA, as a TML-CGA transaction must wait for the writing transaction to commit before it can begin.

$$h_2 = \mathbf{TXBegin}_0 \cdot \mathbf{TXWrite}_0(x, 0) \cdot \mathbf{TXBegin}_1$$

Thus NORec-CGA and TML-CGA are distinct. Histories h_3 and h_4 below demonstrate that the NORec-CGA and NORec2-CGA are distinct.

$$\begin{aligned} h_3 = {} & \mathbf{TXBegin}_0 \cdot \mathbf{TXWrite}_0(x, 1) \cdot \langle \mathtt{TXCommit}_0 \rangle \cdot \mathbf{TXBegin}_1 \cdot \\ & \mathbf{TXRead}_1(x, 0) \cdot \langle \mathtt{TXRead}_1(x), \mathtt{TXAbort}_1 \rangle \\ h_4 = {} & \mathbf{TXBegin}_0 \cdot \mathbf{TXWrite}_0(x, 1) \cdot \langle \mathtt{TXCommit}_0 \rangle \cdot \mathbf{TXBegin}_1 \cdot \\ & \langle \mathtt{TXRead}_1(x), \overline{\mathtt{TXCommit}}_0, \overline{\mathtt{TXRead}}_1(0) \rangle \cdot \mathbf{TXRead}_1(x, 0) \end{aligned}$$

In h_3, transaction 1 reads x and then reads x again, aborting the second read because another writer (transaction 0) has committed. This history is allowed by NORec-CGA but not NORec2-CGA. History h_4 is similar to h_3, but the second read by transaction 0 succeeds because it reads from local read set rather than main memory (which allows it to bypass validation). History h_4 is allowed by NORec2-CGA, but not NORec-CGA. In general, NORec2-CGA aborts strictly less often than NORec-CGA as it contains fewer code paths that lead to an abort occurring. □

Finally, we conclude that the three algorithms we have considered are pairwise distinct, i.e., that the three algorithms considered indeed implement different TM designs.

Lemma 25. TML*,* NORec *and* NORec2 *are pairwise distinct.*

Proof. The proof follows from the equivalence between each algorithm and its coarse-grained counterpart (Lemmas 17 and 20) together with Lemma 24. □

Experimental results. Having checked opacity for our variant of NORec, we implemented the algorithm in C using the STAMP benchmarking suite [7] for transactional memory algorithms, modifying an existing NORec algorithm by Diegues *et al* [13]. Contrary to our initial expectations, a naive implementation of NORec2 is actually slower than ordinary NORec, sometimes up to ten times slower. The reasons for this are as follows:

- By avoiding validation for certain reads, we allow writing transactions that would otherwise abort to continue running, wherein they will invariably (and necessarily) abort in the commit due to validation therein. In general, it seems better for a transaction to fail fast rather than waste time in a doomed state. Tests using the STAMP suite indicate that, for many benchmarks using this variant of NORec, there are millions of transactions performing unnecessary work before they eventually abort when they commit.

- In software TM, the difference in performance between validating the read set (increasing contention to main memory) compared to checking membership of the read set (which is often implemented as a list) is not significant enough to warrant the existence of the additional fast-path check.

We note however that we have shown opacity for transactions that return values from their read set whenever possible. Any implementation of NORec2-CGA where we allow re-reading of values from the read-set only in certain circumstances, e.g., in read-only transactions, and after the last write in a transaction are still valid. This avoids the first performance issue above. Such variants between NORec and NORec2 are clearly opaque. We have benchmarked both variants, and shown that both perform as well as standard NORec, yet transactions abort less often, and therefore represent meaningful (albeit small optimisations) over NORec.

Overall, our method has allowed us to identify an optimisation at a high level of abstraction and rapidly verify opacity of a variant of the NORec algorithm. Modifying the opacity proof and model checking the result took no more than a days worth of work. However, this also demonstrates the importance of concrete benchmarks to evaluate the usefulness of such modifications.

8 Conclusions

Related work. With the widening applicability of TM, there has been an increased interest in verifying opacity of the underlying algorithms. There are direct proofs [12] as well as proofs by simulation [27] that make use the intermediate specification TMS2 [15, 28]. A detailed comparison with this existing (direct) simulation-based proof against TMS2 [28] is given in Section 5. Further comparisons against Derrick *et al.*'s method [12] are give below. Anand *et al.* present a proof method for *conflict opacity*, which is a subclass of opacity [2], however, these proofs are not mechanised. Lesani has developed an alternative criteria, *markability* [29, 26] that allows one to prove opacity by checking that certain conflict ordering properties are satisfied. The markability technique requires reasoning about two orders: an *effect order* relating transactions, and an *observation order*, relating writes and reads. Using the technique involves checking consistency conditions on both these orders, rather than the simple real-time order of linearizability. Another conflict-based technique has also been developed by Guerraoui *et al.* [19, 18, 20] allowing one to reduce a proof of opacity to checking opacity of a system with only two threads and two variables. This reduction depends on the algorithm in question satisfying a number of properties. One point of difference compared with our methods is that Guerraoui *et al.*'s conflict relations do not check the values within each address.

Contributions. Our main contributions for this paper are as follows.

1. We have developed a complete method for reducing proofs of opacity to the simpler and more widely studied condition linearizability. Soundness is proved via an existing result by Derrick *et al.* [12]. These results bring together the previously disconnected worlds of linearizability and opacity verification, and allows one to reuse the vast literature on linearizability verification [16], as well as the growing literature on opacity verification (to verify the coarse-grained abstractions).

2. We have demonstrated our technique using the TML algorithm and shown that the method extends to more complex algorithms by verifying the NORec algorithm. As part of this

verification, we discover a variation of the TMS2 specification, TMS3 that does not require validation when read-only transactions commit. We show that TMS2 is equivalent to TMS3.

3. We have shown that the method naturally copes with relaxed memory by verifying both TML and NORec are opaque under TSO, and neither requires introduction of additional fence instructions. These examples show that a single coarse-grained abstraction is valid for more than one implementation.

4. We developed a variation of NORec which allows reads to be re-read from the read set, and demonstrated that this variation aborts less often than the existing NORec algorithm. Consideration of opacity at higher levels of abstraction elucidates design elements that are not immediately apparent at the level of concrete code. We were able to quickly verify and test this modified algorithm. Through benchmarking, we developed variations with meaningful optimisation, that demonstrate improvements to the original implementation in specific circumstances.

Verifying opacity directly is difficult — because a read-only transaction may serialise against earlier versions of memory, opacity requires one to keep track of all stores created by committed writer transactions. In comparison, linearizability only requires one to keep track of the latest memory snapshot, which simplifies the proof of an implementation. From a verification perspective, a coarse-grained abstraction is straightforward to construct.

The basic idea of linearizing fine-grained operations to verify opacity also appears in [12], where the TML algorithm has been verified. However, their coarse-grained abstraction is very different from ours. Namely, the operations only modify memory and does not utilise any synchronisation code. As a result, their proof must couple an abstraction with the original TML implementation and introduce an explicit history variable. The proof proceeds via induction on the histories generated by TML — these show that each step of TML preserves opacity. However, it is unclear if such an inductive method could scale to more complex algorithms such as NORec, or to include relaxed memory algorithms.

In contrast, our proofs achieve a clear separation of concerns between opacity (of a coarse-grained abstraction) and linearizability (of an implementation), i.e., opacity of a coarse-grained abstraction is verified independently. This separation enables one to distinguish design elements of opaque algorithms at a higher level of abstraction, separating design aspects from synchronisation elements in an implementation that ensure atomicity of fine-grained transactional operations. These insights have been used to develop a new variation of NORec with fast-path read-only transactions.

Experiences. Our experiences suggest that our techniques do indeed simplify proofs of opacity (and their mechanisation). We evidence this by verifying several algorithms with relatively little effort. Our completeness result ensures that the technique is always applicable.

Opacity of each coarse-grained abstraction is generally trivial to verify (our proofs are mechanised in Isabelle), leaving one with a proof of linearizability of an implementation against this abstraction. The second step is limited only by techniques for verifying linearizability. We have opted for a model checking approach using PAT, which enables linearizability to be checked by automatically checking refinement. These show that the coarse-grained abstractions we have developed are actually equivalent to their implementations up to some bound on the number of transactions and size of the store. It is of course also possible to perform a full verification of linearizability.

Overall, PAT was well suited for the task of model checking linearizability. We did not need to provide PAT with any information on the linearization points, invariants, or simulation relations between the concrete implementations and CGA abstractions. Once we had encoded our TM algorithms in PAT, verification was fully automatic. The task of implementing the TM algorithms in PAT was made particularly simple due to its support for CSP style processes combined with shared mutable state. Other CSP style model checkers such as FDR3 [17] have a more functional specification language disallowing such mutable state. To implement in our algorithms in such a system would require encoding the shared state as message passing between processes, which would significantly increase the distance between the model and the natural pseudocode implementations of the algorithms, which in PAT, is fairly minimal. We note that while PAT did not appear to effectively take advantage of the 24 cores on the machine we ran our model-checking on, our verifications were limited by memory usage rather than processor speed. For example, model checking TML with four transactions used over 40GB of memory. It is not clear whether this high memory usage could be reduced with another model checker, or if it is inherent in the complexity of the models themselves.

Future work. Our work suggests that to fully verify a TM algorithm using coarse-grained abstraction, the bottleneck to verification is the proof of linearizability itself [16]. It is therefore worthwhile considering whether linearizability proofs can be streamlined for transactional objects. For example, Bouajjani *et al.* have shown that for particular inductively-defined data structures, linearizability can be reduced to state reachability [5]. Exploration of whether such methods apply to transactional objects remains a topic for future work. Establishing this link would be a useful result — it would allow one to further reduce a proof of opacity to a proof of state reachability.

Verifying linearizability sometimes requires information about the possible (future) executions of an algorithm [16, 24, 36]. The examples we have considered have not required consideration of (speculative) future-based behaviour, but such TM algorithms do exist, e.g., TL2 [9]. However, for such algorithms, a direct proof of opacity also requires one to use the same futuristic information [27]. Development of coarse-grained abstractions for algorithms such as TL2 remains a topic of future work — it is worth noting that our completeness result ensures that this can be done.

This work can also contribute to the *characterisation* of algorithms at a high-level of abstraction. That is, by verifying a number of other algorithms from the literature, it will be possible to extend the tree in Figure 1, and understand particular design features at a coarse-grained level.

We thank John Derrick for helpful discussions and funding from EPSRC EP/N016661/1.

References

[1] S. V. Adve and J. K. Aggarwal. A unified formalization of four shared-memory models. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):613–624, June 1993.

[2] A. S. Anand, R. K. Shyamasundar, and S. Peri. Opacity proof for CaPR+ algorithm. In *ICDCN*, ICDCN '16, pages 16:1–16:4, NY, USA, 2016. ACM.

[3] A. Armstrong, B. Dongol, and S. Doherty. Isabelle and PAT theories for reducing opacity to linearizability, 2016. Ancillary material.

[4] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. A programming language perspective on transactional memory consistency. In P. Fatourou and G. Taubenfeld, editors, *PODC '13*, pages 309–318. ACM, 2013.

[5] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. On reducing linearizability to state reachability. In M. M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, editors, *ICALP*, volume 9135 of *LNCS*, pages 95–107. Springer, 2015.

[6] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: a complete and automatic linearizability checker. In *PLDI*, pages 330–340. ACM, 2010.

[7] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In D. Christie, A. Lee, O. Mutlu, and B. G. Zorn, editors, *IISWC 2008*, pages 35–46. IEEE Computer Society, 2008.

[8] P. Cerný, A. Radhakrishna, D. Zufferey, S. Chaudhuri, and R. Alur. Model checking of linearizability of concurrent list implementations. In *CAV*, volume 6174 of *LNCS*, pages 465–479. Springer, 2010.

[9] Dice D, O. Shalev, and N. Shavit. Transactional locking II. In S. Dolev, editor, *DISC*, volume 4167 of *LNCS*, pages 194–208. Springer, 2006.

[10] L. Dalessandro, D. Dice, M. L. Scott, N. Shavit, and M. F. Spear. Transactional mutex locks. In P. D'Ambra, M. R. Guarracino, and D. Talia, editors, *Euro-Par (2)*, volume 6272 of *LNCS*, pages 2–13. Springer, 2010.

[11] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: streamlining STM by abolishing ownership records. In R. Govindarajan, D. A. Padua, and M. W. Hall, editors, *PPoPP*, pages 67–78. ACM, 2010.

[12] J. Derrick, B. Dongol, G. Schellhorn, O. Travkin, and H. Wehrheim. Verifying opacity of a transactional mutex lock. In *FM*, volume 9109 of *LNCS*, pages 161–177. Springer, 2015.

[13] N. Diegues, P. Romano, and L. Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *PACT '14*, pages 3–14, New York, NY, USA, 2014. ACM.

[14] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE*, volume 3235 of *LNCS*, pages 97–114. Springer, 2004.

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

[16] B. Dongol and J. Derrick. Verifying linearisability: A comparative survey. *ACM Comput. Surv.*, 48(2):19, 2015.

[17] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. Fdr3 — a modern refinement checker for csp. In E. Ábrahám and K. Havelund, editors, *TACAS 2014*, pages 187–201, Berlin, Heidelberg, 2014. Springer.

[18] R. Guerraoui, T. A. Henzinger, and V. Singh. Completeness and nondeterminism in model checking transactional memories. In F. van Breugel and M. Chechik, editors, *CONCUR*, pages 21–35. Springer, 2008.

[19] R. Guerraoui, T. A. Henzinger, and V. Singh. Model checking transactional memories. *Distributed Computing*, 22(3):129–145, 2010.

[20] R. Guerraoui, T. A. Henzinger, and V. Singh. Verification of STM on relaxed memory models. *FMSD*, 39(3):297–331, 2011.

[21] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In S. Chatterjee and M. L. Scott, editors, *PPOPP*, pages 175–184. ACM, 2008.

[22] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory.* Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[23] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[24] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

[25] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[26] M. Lesani. *On the Correctness of Transactional Memory Algorithms.* PhD thesis, UCLA, 2014.

[27] M. Lesani, V. Luchangco, and M. Moir. A framework for formally verifying software transactional memory algorithms. In M. Koutny and I. Ulidowski, editors, *CONCUR 2012*, pages 516–530. Springer Berlin Heidelberg, 2012.

[28] M. Lesani, V. Luchangco, and M. Moir. Putting opacity in its place. In *Workshop on the Theory of Transactional Memory*, 2012.

[29] M. Lesani and J. Palsberg. Decomposing opacity. In F. Kuhn, editor, *DISC*, volume 8784 of *LNCS*, pages 391–405. Springer, 2014.

[30] Y. Liu, W. Chen, Y. A. Liu, J. Sun, S. Jie Zhang, and J. Song Dong. Verifying linearizability via optimized refinement checking. *IEEE Trans. Software Eng.*, 39(7):1018–1039, 2013.

[31] N. Lynch and F. Vaandrager. Forward and backward simulations. *Information and Computation*, 121(2):214 – 233, 1995.

[32] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[33] O. Müller. I/O Automata and beyond: Temporal logic and abstraction in Isabelle. In J. Grundy and M. Newey, editors, *TPHOLs*, pages 331–348. Springer Berlin Heidelberg, 1998.

[34] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS.* Springer, 2002.

[35] S. Owens. Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In Theo DHondt, editor, *ECOOP 2010*, volume 6183 of *LNCS*, pages 478–503. Springer Berlin Heidelberg, 2010.

[36] G. Schellhorn, J. Derrick, and H. Wehrheim. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. Comput. Log.*, 15(4):31:1–31:37, 2014.

[37] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: towards flexible verification under fairness. In *CAV*, volume 5643 of *LNCS*, pages 709–714. Springer, 2009.

[38] V. Vafeiadis. Automatically proving linearizability. In *CAV*, volume 6174 of *LNCS*, pages 450–464. Springer, 2010.

[39] M. T. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In *SPIN*, volume 5578 of *LNCS*, pages 261–278. Springer, 2009.