On the Correctness of Transactional Memory

Rachid Guerraoui Michał Kapałka

School of Computer and Communication Sciences, EPFL

{rachid.guerraoui, michal.kapalka}@epfl.ch

Abstract

Transactional memory (TM) is perceived as an appealing alternative to critical sections for general purpose concurrent programming. Despite the large amount of recent work on TM implementations, however, very little effort has been devoted to precisely defining what guarantees these implementations should provide. A formal description of such guarantees is necessary in order to check the correctness of TM systems, as well as to establish TM optimality results and inherent trade-offs.

This paper presents *opacity*, a candidate correctness criterion for TM implementations. We define opacity as a property of concurrent transaction histories and give its graph theoretical interpretation. Opacity captures precisely the correctness requirements that have been intuitively described by many TM designers. Most TM systems we know of do ensure opacity.

At a very first approximation, opacity can be viewed as an extension of the classical database serializability property with the additional requirement that even *non-committed* transactions are prevented from accessing inconsistent states. Capturing this requirement precisely, in the context of general objects, and without precluding pragmatic strategies that are often used by modern TM implementations, such as versioning, invisible reads, lazy updates, and open nesting, is not trivial.

As a use case of opacity, we prove the first lower bound on the complexity of TM implementations. Basically, we show that every single-version TM system that uses invisible reads and does not abort non-conflicting transactions requires, in the worst case, $\Omega(k)$ steps for an operation to terminate, where k is the total number of objects shared by transactions. This (tight) bound precisely captures an inherent trade-off in the design of TM systems. The bound also highlights a fundamental gap between systems in which transactions can be fully isolated from the outside environment, e.g., databases or certain specialized transactional languages, and systems that lack such isolation capabilities, e.g., general TM frameworks.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.4 [*Software Engineering*]: Software/Program Verification; F.2.2 [*Analysis of Algorithms and Problem Complexity*]: Nonnumerical Algorithms and Problems

General Terms Theory, Verification

Keywords Transactional memory, model, correctness, lower bound

1. Introduction

Transactional memory (TM) [15, 28] is a programming paradigm in which concurrent threads synchronize via in-memory *transactions*. A transaction is an explicitly delimited sequence of operations on shared objects. Transactions are *atomic*: programmers get the illusion that every transaction is executed *instantaneously*, at some single, unique point in time, and does not observe any concurrency from other transactions. The changes performed by a transaction on shared objects are immediately visible (to other transactions) if the transaction commits, and are completely discarded if the transaction aborts.

The TM paradigm has raised a lot of hope for mastering the complexity of concurrent programming. The aim is to provide the programmer with an abstraction, i.e., the transaction [8], that makes concurrency as easy as with coarse-grained critical sections, while exploiting the underlying multi-core architectures as well as handcrafted fine-grained locking, which is difficult and error-prone. It is thus not surprising to see a large body of work directed at experimenting with various kinds of TM implementation strategies, e.g. [15, 28, 14, 13, 18, 5, 19, 12, 29, 25]. What might be surprising is the little formalization of the *precise* guarantees that TM implementations should provide. Without such formalization, it is impossible to check the correctness of these implementations, establish any optimality result, or determine whether TM design trade-offs are indeed fundamental or simply artifacts of certain environments.

From a user's perspective, a TM should provide the same semantics as critical sections: transactions should appear as if they were executed sequentially. However, a TM implementation would be inefficient if it never allowed different transactions to run concurrently. Reasoning about the correctness of a TM implementation goes through defining a way to state precisely whether a given execution in which a number of transactions execute steps in parallel "looks like" an execution in which these transactions proceed one after the other. The role of a correctness criterion in this context is precisely to capture what the very notion of "looks like" really means.

At first glance, it seems very likely that such a criterion would correspond to one of the numerous ones defined in the literature, e.g., linearizability [16], serializability [24, 2], rigorous scheduling [4], etc. We argue, however, that none of these criteria, nor any straightforward combination or extension thereof, is sufficient to describe the semantics of TM with its subtleties. In particular, none of them captures exactly the very requirement that every transaction, including a *live* (i.e., not yet completed) one, accesses a *consistent* state, i.e., a state produced by a sequence of previously committed transactions. While a live transaction that accesses an inconsistent state can be rendered harmless in database systems simply by being aborted, such a transaction might create significant dangers when executed within a general TM framework, as we illustrate later in this paper. It is thus not surprising that most TM implementations employ mechanisms that disallow such situations,

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.
Copyright © 2008 ACM 978-1-59593-960-9/08/0002…$5.00

sometimes at a big cost. At a very high level, disallowing transactions to access inconsistent states resembles, in the database terminology, preventing *dirty reads* or, more generally, the *read skew* phenomenon [1], when generalized to all transactions (not only committed ones as in [1]) and arbitrary objects.

In this paper, we present *opacity*, a correctness criterion aimed at capturing the semantics of TM systems. The technical challenge in specifying opacity is the ability to reason about states accessed by live transactions, and to do so in a model (a) with arbitrary objects, beyond simple read/write variables, (b) possibly with multiple versions of each object, and (c) without precluding various TM strategies and optimization techniques, such as invisible reads, lazy updates, caching, or open nesting.

Most transactional memory systems we know of ensure opacity, including DSTM [14], ASTM [18], SXM [13], JVSTM [5], TL2 [6], LSA-STM [25] and RSTM [19]. They do so by combining classical database concurrency and recovery control schemes with additional validation strategies, which ensure that *every* return value of an operation executed by a transaction is consistent with the return values of all previous operations of the very same transaction. (This leads to aborting the transaction if there is any risk of accessing an inconsistent state.) These strategies are usually implemented using the single-writer multiple-readers pattern, with either explicit locks (e.g., TL2) or "virtual", revocable ones (e.g., obstruction-free TMs, such as DSTM, ASTM and SXM), sometimes with a multi-versioning scheme (e.g., LSA-STM and JVSTM) or specialized optimization strategies.

There are indeed TM implementations that do not ensure opacity; these, however, explicitly trade safety guarantees, while recognizing the resulting dangers, for improved performance. Examples are: a version of SI-STM [26] and the TM described in [7]. We believe that opacity can also be used as a reference point for expressing the semantics of such TM implementations and deriving other, possibly weaker, correctness criteria. This would enable fair comparison between TM algorithms and better recognition of their safety-performance trade-offs.

Besides defining opacity, we also present its graph characterization. Basically, we show how to build a graph that visualizes dependencies between transactions in a given execution, and how to express opacity in terms of acyclicity of such a graph. This interpretation helps proving correctness of TM implementations, highlighting opacity of a given execution, or visualizing opacity violations.

As a use case for opacity, we establish the first complexity bound for TM implementations. Roughly speaking, we prove that TM implementations that ensure opacity while (1) using invisible reads,[1] (2) ensuring that no transaction is aborted unless it conflicts with another live transaction, and (3) employing a single-version scheme, require, in the worst case, $\Omega(k)$ steps for per-operation *validation*, where k is the total number of objects shared by transactions.

This lower bound is tight: DSTM and ASTM ensure opacity and have the above three properties, and require, in the worst case, $\Theta(k)$ steps to complete a single operation (or, in other words, $\Theta(k^2)$ steps to execute a transaction that accesses k objects). On the other hand, TM implementations that use visible reads, e.g., SXM and RSTM, or abort transactions more often, e.g., TL2, can have a constant complexity.[2]

Indirectly, the lower bound also highlights a gap between *database transactions*, or, more generally, systems that support full isolation of transactional code from the outside environment, for which serializability is sufficient, and *memory transactions* (in the sense of most TM frameworks). Indeed, our bound does not hold for serializability, even when considered in its strict form to account for real-time order and combined with recoverability [11]. In this sense, requiring opacity is a key to establish our lower bound and hence capture the trade-off between implementations like DSTM and ASTM on one hand, and implementations like SXM, RSTM or TL2 on the other hand.

To summarize, this paper contributes to the study of transactional memory systems: we present (a) a candidate correctness criterion to measure the correctness of a TM implementation, together with its graph characterization, and (b) the first lower bound on the complexity of TM implementations.

The rest of the paper is organized as follows. We first give an intuitive description of what is generally expected from a TM and argue why a new correctness criterion is indeed necessary to capture this intuition. We then define our notion of opacity and describe its graph characterization. Next, we establish our complexity lower bound. We conclude by discussing complementary issues such as how one can deal with mixing transactional and non-transactional operations [3], encompass nested transactions [20, 22], or specify progress properties [27].

Due to space limitations, we give here only an intuition behind the lower bound result. A complete proof of the lower bound, precise definitions of the terms used thereof, as well as a proof of correctness of the graph characterization of opacity, can be found in the full version of this paper [10].

2. Expectations

Nearly every paper about TM gives some intuition about what a TM implementation should ensure. Clearly, committed transactions should appear as if they executed instantaneously, at some unique point in time, and aborted transactions, as if they did not execute at all. Additionally, the following two guarantees (both provided by critical sections) are considered (sometimes implicitly) as essential aspects of TM semantics.

Preserving real-time order. It is generally required from a TM that the point in time at which a transaction appears to occur lies somewhere within the lifespan of the transaction. This means that a transaction should not observe an outdated state of the system, which can be the case if extensive caching of object states is used. That is, if a transaction T_1 modifies an object x and commits, and then another transaction T_2 starts and reads x, then T_2 should read the value written by T_1 and not an older value. More generally, if a transaction T_i commits before a transaction T_j starts, then T_i should indeed appear as if it executed before T_j.

Violating real-time ordering may lead to counter-intuitive situations, as explained in [24], and mislead programmers typically used to critical sections that naturally enforce real-time ordering. Preserving real-time ordering is also particularly important when transactions can read from (or write to) devices that are not controlled by the TM, e.g., clocks or storage devices.

Precluding inconsistent views. A more subtle issue is related to the state accessed by *live* transactions (i.e., transactions that did not commit or abort yet). Because a live transaction can always be later aborted, and its updates discarded, one might simply assume that the remedy to a transaction that accesses an inconsistent state is to abort it. This is the case for databases, in which transactions are executed in a fully controlled environment. However, memory transactions are autonomous programs. As argued in [29], a transaction

[1] With *invisible reads*, no process knows about read operations issued by transactions executed by *other* processes. Several TM implementations optimize their performance with invisible reads, e.g. DSTM, ASTM, and TL2.

[2] For multi-version TM implementations, like LSA-STM or JVSTM, the complexity is not constant. However, it can be bounded by a function independent of k.

that accesses an inconsistent state can cause various problems, even if it is later aborted.

To illustrate this, consider two shared objects, x and y. A programmer may assume that y is always equal to x^2, and $x \geq 2$. Clearly, the programmer will then take care that every transaction, when executed as a whole, preserves the assumed invariants. Assume the initial value of x and y is 4 and 16, respectively, and let T_1 be a transaction that performs the following operations:

```
x := 2; y := 4; commit
```

Now, if another transaction T_2 executes concurrently with T_1 and reads the old value of x (4) and the new value of y (also 4), the following problems may occur, even if T_2 is to be aborted later: First, if T_2 tries to compute the value of $1/(y - x)$, then a "divide by zero" exception will be thrown, which can crash the process executing the transaction or even the whole application. Second, if T_2 enters the following loop:

```
t := x
do array[t] := 0; t := t + 1
until t = y
```

then unexpected memory locations could be overwritten, not to mention that the loop would need to span the entire value domain.[3] Other examples [29] include situations where a transaction that observes an inconsistent state performs direct (and unexpected) IO operations, which are difficult to undo and thus usually forbidden within transactions.

When programs are run in managed environments, these problems can be solved by carefully isolating transactions from the outside world (sandboxing), as in databases. However, it is commonly argued that sandboxing is expensive and applicable only to specific run-time environments [6].[4]

3. Why a New Correctness Criterion for TM?

Given the large body of literature on concurrency control, it seems a priori very likely that the intuition behind TM semantics is already captured by some existing consistency criterion. We argue below that this is not the case.

3.1 Linearizability

Linearizability [16], a safety property devised to describe shared objects, is sometimes used as a correctness criterion for TM. In the TM terminology, linearizability means that, intuitively, every transaction should appear as if it took place at some single, unique point in time during its lifespan. Clearly, aborted transactions have to be accounted for, e.g., through an extension of linearizability described in [31].

Linearizability would be an appropriate correctness criterion for TM if transactions were external to the application using them, i.e., if only the end result of a transaction counted. However, a TM transaction is not a black box operation on some complex shared object but an internal part of an application: the result of every operation performed inside a transaction is important and accessible to a user. As indicated in the original paper on linearizability [16], serializability and its derivatives are more suitable a base to reason about the correctness of transaction executions.

3.2 Serializability

Serializability [24] is one of the most commonly required properties of database transactions. Roughly speaking, a history H of transactions (i.e., the sequence of operations performed by all transactions in a given execution) is serializable if all *committed* transactions in H issue the same operations and receive the same responses as in some *sequential* history S that consists only of the transactions committed in H. (A sequential history is, intuitively, one with no concurrency between transactions.)

Serializability, even considered in its *strict* form [24] to account for real-time ordering, is not sufficient for modelling a TM for various reasons: (a) it relies on the implicit assumption that a read operation on a shared object x always returns the last value previously written to x; (b) it is restricted only to read and write operations, and (c) it does not say anything about the state accessed by live (or aborted) transactions. As we discuss below, variants of serializability tackle some of these issues but none of them, nor any clear combination thereof, does the entire job.

3.3 1-Copy Serializability

Memory transactions may create local or shared copies of some shared objects and use them temporarily for their operations. Thus, a transaction T_i, when reading a shared object x, may be returned one of the many versions of x that are globally or locally accessible to T_i, not necessarily the most recent one.

1-copy serializability [2] is similar to serializability, but allows for multiple versions of any shared object, while giving the user an illusion that, at any given time, only one copy of each shared object is accessible to transactions. Besides not requiring anything about the state accessed by live transactions, a major limitation of 1-copy serializability is the underlying model being restricted only to read and write operations.

3.4 Global Atomicity

It is usually argued that providing shared objects with richer semantics than simple read-write variables can decrease the probability of conflicts between transactions and thus increase throughput [22, 23]. To illustrate this, consider several transactions concurrently increasing a counter x, without reading its value:

```
T1:       T2:       ...   Tk:
x.inc()   x.inc()         x.inc()
commit    commit          commit
```

In a system that supports only read and write operations, each transaction has to first read x and then write a new value to x. Unfortunately, among the transactions that read the same value from x, only one can commit (otherwise, (1-copy) serializability is violated). Clearly, when the system recognizes the semantics of the `inc` operation, there is no reason why the transactions could not proceed and commit concurrently. More generally, a TM implementation may exploit the benefits of operations that are idempotent, commutative, or write-only (see [22] for more elaborate examples).

Supporting arbitrary shared objects brings, however, additional significant difficulties in reasoning about correctness. We can no longer assume that each operation is either read-only or write-only, and that each shared object is historyless, or even deterministic (in the most general case). We need to consider a formal description of the semantics of the implemented shared objects as an input parameter to the TM correctness criterion, not as its integral part. A further complication comes from the fact that certain operations cannot be undone. Some TM implementations might allow such operations to be executed by a transaction, e.g., by buffering them until the transaction is guaranteed to commit and speculating on return values. Thus, we cannot include roll-back operations in a history to model aborted transactions.

[3] Note that this situation does not necessarily result in a "segmentation fault" signal that is usually easy to catch. Basically, the loop may overwrite memory locations of variables that belong to the application executing the loop but are outside control of the TM implementation.

[4] Sandboxing would for instance be difficult to achieve for applications written in low-level languages (like C) and executed directly by an operating system.

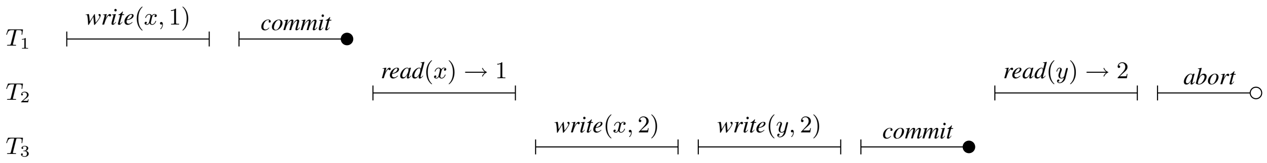

Figure 1. A history that satisfies global atomicity (with real-time ordering guarantees) and recoverability, but in which an aborted transaction (T_2) accesses an inconsistent state of the system (x and y are simple variables/objects that implement *read* and *write* operations)

Global atomicity [30] is a general form of serializability that addresses the above issue. It (a) is not restricted only to read-write objects, and (b) does not preclude several versions of the same shared object. Nevertheless, global atomicity restricts only the execution of committed transactions and does not require anything about the state accessed by live (or aborted) transactions. Therefore, it needs to be extended accordingly.

3.5 Recoverability

Recoverability [11] puts restrictions on the state accessed by *every* transaction, including a live one. Intuitively, recoverability precludes certain undesirable effects, such as cascading aborts, which may occur when a live transaction observes changes done by another live transaction. In its strongest form, recoverability requires, intuitively, that if a transaction T_i updates a shared object x, then no other transaction can perform an operation on x until T_i commits or aborts. It may seem at first that recoverability, combined with global atomicity, and extended to account for real-time ordering of transactions, matches the TM requirements highlighted in Section 2. Unfortunately, this is not the case, as illustrated by the following example.

Consider a history H corresponding to the scenario depicted in Figure 1. H satisfies global atomicity: transaction T_2 aborts and transactions T_1 and T_3 are sequential. Moreover, H satisfies recoverability: T_2 accesses x after T_1 commits and before T_3 starts, whilst T_2 accesses y after T_3 commits. Nevertheless, T_2 accesses an inconsistent state: T_2 could not have read $x = 1$ and $y = 2$ if T_2 was executed between T_1 and T_3, or after T_3.

On the other hand, recoverability restricts TM implementations too much in a general model with arbitrary shared objects. For instance, consider the example from Section 3.4 in which many transactions try to increment a shared counter. Recoverability does not allow them to proceed concurrently, for each modifies the same shared object. However, there is no reason why a TM implementation could not execute them in parallel: even if one of these transactions aborts, it has no influence on the others (at least as long as no transaction reads the value of the counter).

3.6 Rigorous Scheduling

At a high level, what seems to be required is a correctness criterion precluding any two transactions from concurrently accessing an object if one of them updates that object. Restricted to read-write objects (registers), this resembles the notion of rigorous scheduling [4] in database systems. As we argue through the following example, however, this would be too strong and would preclude valid TM implementations.

Consider the following situation in which several transactions concurrently update overlapping sets of objects:

```
T1:      T2:      ...   Tk:
x := 1   x := 2         x := k
y := 1   y := 2         y := k
z := 1   z := 2         z := k
commit   commit         commit
```

Rigorous scheduling requires that all but one of the transactions get blocked or aborted. However, a user does not really care as long as the end result is consistent (i.e., reading x, y and z always gives $x = y = z \in \{1, \ldots, k\}$). We can imagine a TM implementation that executes the write operations in a "smart" way (e.g., making sure that some transactions do not overwrite results of other ones) and thus allows for more concurrency. Such an implementation could be fine from a user's perspective, and so should not be considered incorrect.

3.7 Towards Extending Global Atomicity

In short, formalizing the TM semantics goes through finding a way to extend global atomicity with the requirement that live (and aborted) transactions always access consistent state, but without limiting the generality of the model. This is not trivial, mostly for the following two reasons. First, because we consider arbitrary objects' operations, some of which cannot always be undone, we are not able to model aborted transactions by simply inserting "virtual" events that roll-back the changes done by these transactions.

Second, a user's application commits a transaction by submitting a commit request to a TM implementation and waiting for the response. Thus, there is no single commit event, unlike in database models: the transaction gets committed somewhere between the request and the response events. Even TM implementations do not always commit transactions in a single step. While this looks like a minor detail, it has important implications. Basically, a live transaction for which a commit request has been issued can appear as committed or aborted depending on the context. Thus, expressing the semantics of live transactions is a challenging problem.

4. Model of Transactional Memory

Before describing our new correctness criterion, we introduce here a precise model of a TM as seen from a user's perspective. The formalism given here underlies our notion of opacity, but is general enough to be a base for other, possibly weaker, correctness criteria or alternative properties. In Section 6, we will extend the model given here to include operations (e.g., hardware instructions) used by software TM implementations.[5]

Our model is similar to the one in [30]. The main difference is the way we treat the termination of transactions, which is crucial in the TM context: We consider a pair of commit-try and commit events instead of a single atomic commit step (cf. Section 3.7). Besides, we define additional terms related to live transactions, which are used for specifying opacity.

Transactions and shared objects. A TM allows for threads of an application to communicate by executing *transactions*. A transaction may perform *operations* on *shared objects*, as well as local computations on objects inaccessible to other transactions. An operation (on a shared object) may take some arguments and return some value. We denote by *Obj* the set of objects shared by transactions.

[5] Software TM implementations provide TM semantics to a user's application in systems that do not support memory transactions in hardware.

Every shared object exports a certain set of operations. For example, a *register* object (which is often used in the examples in this paper) exports operations *read* and *write*. The *read* operation takes no arguments (or an empty argument $\perp$) and returns the current state of the register. The $\mathit{write}(v)$ operation sets the state of the register to the value v given as an argument and always returns *ok*. (Clearly, the domain of possible values of v will be restricted in most cases.)

Every transaction has a unique *identifier* from a set $\mathit{Trans} = \{T_1, T_2, \ldots\}$. Every transaction is initially *live* and may eventually become either *committed* or *aborted*, as explained in detail in the following paragraphs. A transaction that is not live does no longer perform any actions. Retrying an aborted transaction (i.e., the computation the transaction intends to perform) is considered in our model as a new transaction, with a different transaction identifier.

Transactional events. In order to execute an operation *op* on a shared object ob, a transaction T_i (i.e., a transaction with identifier T_i) issues an *operation invocation* event $\mathit{inv}_i(ob, op, \mathit{args})$ and expects a matching *operation response* event $\mathit{ret}_i(ob, op, \mathit{val})$, where *args* are the arguments passed to the operation and *val* is the value returned by the operation. A transaction is sequential, in the sense that it does not invoke any operation until it receives a response from the last operation it invoked. An operation invocation event and an operation response event *match* if they are issued by/for the same transaction and refer to the same shared object and operation.

A transaction T_i might also issue two special events: a *commit-try* event tryC_i or an *abort-try* event tryA_i. After issuing tryC_i or tryA_i, transaction T_i waits for a *commit* event C_i or an *abort* event A_i. Intuitively, tryC_i expresses the will of transaction T_i to commit. In response, the transaction can get either committed (event C_i) or aborted (event A_i). An event tryA_i indicates that transaction T_i wants to be aborted and always results in an abort event A_i for T_i.[6] A commit-try/abort-try event and a commit/abort event *match* if they are issued by/for the same transaction.

An abort event might also be received by a transaction instead of an operation response event. This usually happens if the TM knows that the transaction will not be able to commit later (because of conflicts with other transactions), or if the TM cannot return an operation response event with no risk of violating opacity.

We divide events into two categories. Operation invocation, commit-try and abort-try events are called *invocation events*. Operation response, commit and abort events are called *response events*. Invocation events are initiated by transactions, and response events—by a TM. As every transaction is an integral part of an application, and is fully controlled by its application thread, a TM does not know in advance which invocation events will be issued by a transaction. That is, the TM does not know which operations on which shared objects a transaction will perform, and whether the transaction will request to be committed (commit-try event) or aborted (abort-try event).

An *operation execution* is a pair of an operation invocation event and a matching operation response event. That is, an operation execution $\mathit{exec}_i(ob, op, \mathit{args}, \mathit{val})$ is a sequence $\langle \mathit{inv}_i(ob, op, \mathit{args}), \mathit{ret}_i(ob, op, \mathit{val})\rangle$.[7] When there is no ambiguity, we will say *operation* and *operation execution* interchangeably.

When considering register objects, we use the following simplified notation. We denote by $\mathit{read}_i(r, v)$ a *read* operation execution on register r, by transaction T_i, returning value v, and by $\mathit{write}_i(r, v)$ a *write* operation execution on register r, by T_i, with value v given as an argument. More formally, $\mathit{read}_i(r, v) = \mathit{exec}_i(r, \mathit{read}, \perp, v)$, and $\mathit{write}_i(r, v) = \mathit{exec}_i(r, \mathit{write}, v, \mathit{ok})$.

Transaction histories. A (high-level) *history* is the sequence of all invocation and response events that were issued and received by transactions in a given execution.[8] Thus, we assume that all events of an execution can be totally ordered according to the time at which they were issued. Simultaneous events (e.g., on multiprocessor systems) can be ordered arbitrarily.

We use the following notations. Consider any history H:

- $H|T_i$ denotes the longest subsequence of history H that contains only events executed by transaction T_i,
- $H|ob$ denotes the longest subsequence of history H that contains only operation invocation events and operation response events on shared object ob, and
- $H \cdot H'$ denotes the concatenation of histories H and H'.

We say that a transaction T_i is in history H, and write $T_i \in H$, if $H|T_i$ is a non-empty sequence, i.e., if there is at least one event of T_i in H.

We assume that every history H is *well-formed*. Intuitively, this means that the sequence of events at *each individual* transaction T_i (i.e., the history $H|T_i$) is of the form: an invocation event, a matching response event, an invocation event, and so on, where (1) no event follows a commit or abort event, (2) only a commit or abort event can follow a commit-try event, and (3) only an abort event can follow an abort-try event. More formally, for every transaction $T_i \in \mathit{Trans}$, history $H|T_i$ is a prefix of a sequence $O \cdot F$, where O is a sequence of operation executions issued by transaction T_i, and F is one of the following sequences: (1) $\langle \mathit{inv}_i(ob, op, \mathit{args}), A_i\rangle$ (for some shared object ob, an operation *op* of ob, and arguments *args* of *op*), (2) $\langle \mathit{tryA}_i, A_i\rangle$, (3) $\langle \mathit{tryC}_i, C_i\rangle$, or (4) $\langle \mathit{tryC}_i, A_i\rangle$.

Intuitively, we consider two histories to be *equivalent*, if they contain the same transactions, and every transaction issues the same invocation events and receives the same response events in both histories. Thus, equivalent histories differ only in the relative position of events of different transactions. More precisely, we say that histories H and H' are equivalent, and write $H \equiv H'$, if, for every transaction $T_i \in \mathit{Trans}$, $H|T_i = H'|T_i$.

We say that an invocation event e issued by a transaction T_i is *pending* in a history H, if there is no response event matching e and following e in history $H|T_i$.

For example, the following (well-formed) history H_1 corresponds to the execution depicted in Figure 1:

$$\begin{aligned} H_1 \quad = \quad & \langle \mathit{write}_1(x,1), \mathit{tryC}_1, C_1, \mathit{read}_2(x,1), \\ & \mathit{write}_3(x,2), \mathit{write}_3(y,2), \mathit{tryC}_3, C_3, \\ & \mathit{read}_2(y,2), \mathit{tryC}_2, A_2\rangle. \end{aligned}$$

Clearly, there is no pending invocation event in H_1. The following history H_2 is one of the histories that are equivalent to H_1:

$$\begin{aligned} H_2 \quad = \quad & \langle \mathit{write}_1(x,1), \mathit{tryC}_1, C_1, \\ & \mathit{write}_3(x,2), \mathit{write}_3(y,2), \mathit{tryC}_3, C_3, \\ & \mathit{read}_2(x,1), \mathit{read}_2(y,2), \mathit{tryC}_2, A_2\rangle. \end{aligned}$$

Status of transactions. If the last event of a transaction T_i in a history H is C_i or A_i, then we say that T_i is, respectively, *committed* or *aborted* in H. A transaction that is committed or aborted is *completed*. A transaction that is not completed is called *live*. An aborted transaction that did not issue an abort-try event is said to be *forcefully aborted*. A live transaction that has issued a commit-try event is said to be *commit-pending*.

[6] We could alternatively let a transaction issue an abort event directly, but then it would be difficult to distinguish the case in which a transaction aborts itself voluntarily from the case in which the transaction is aborted by the TM implementation (e.g., upon an unresolvable conflict).

[7] We denote by $\langle e_1, \ldots, e_k\rangle$ the sequence of events $e_1, \ldots, e_k$.

[8] Note that a history includes only *transactional* events, i.e., the events described in the previous paragraphs of this section.

For example, in history H_1 described before, all transactions are completed. Transactions T_1 and T_3 are committed in H_1, while transaction T_2 is forcefully aborted in H_1.

Real-time order of transactions. There is a clear happen-before relation between a completed transaction T_i and every transaction that issues its first event after T_i becomes committed or aborted (in a given history H). This happen-before relation in a history H, which we denote by $\prec_H$, defines what we call the *real-time order* of transactions in H. More precisely, for every history H, relation $\prec_H$ is the partial order on the transactions in H, such that, for any two transactions $T_i, T_j \in H$, if T_i is completed and the first event of T_j follows the last event of T_i in H, then $T_i \prec_H T_j$.

We say that transactions $T_i, T_j \in H$ are *concurrent* in history H if they are not ordered by the happen-before relation $\prec_H$, i.e., if $T_i \not\prec_H T_j$ and $T_j \not\prec_H T_i$.

We say that a history H' *preserves the real-time order* of a history H, if $\prec_H \subseteq \prec_{H'}$. That is, if $T_i \prec_H T_j$, then $T_i \prec_{H'} T_j$, for any two transactions T_i and T_j in H.

For example, consider history H_1 described before. In H_1, transactions T_2 and T_3 are concurrent, $T_1 \prec_{H_1} T_2$, and $T_1 \prec_{H_1} T_3$. Any history H for which $T_1 \prec_H T_2$ and $T_1 \prec_H T_3$ (e.g., history H_2) preserves the real time order of H_1.

Sequential histories. A (well-formed) history H is *sequential* if no two transactions in H are concurrent. Sequential histories are of special interest, because their correctness is trivial to verify, given a precise semantics of the shared objects and their operations.

For example, history H_2 introduced before is sequential. On the contrary, history H_1 (equivalent to H_2) is not sequential, because transactions T_2 and T_3 are concurrent in H_1.

Complete histories. We say that a history H is *complete* if H does not contain any live transaction. For example, histories H_1 and H_2 used in the previous examples are both complete.

If a history H is not complete, then we can transform it to a complete history H' by aborting or committing the live transactions in H. More specifically, for every history H we define a set of (well-formed) histories $\mathit{Complete}(H)$. Intuitively, every history H' in $\mathit{Complete}(H)$ is obtained from history H by committing or aborting every commit-pending transaction in H, and aborting every other live transaction in H. More precisely, a history H' is in $\mathit{Complete}(H)$, if (1) H' is well-formed, (2) H' is obtained from H by inserting a number of commit-try, commit and abort events for transactions that are live in H, (3) every transaction that is live and not commit-pending in H is aborted in H', and (4) every transaction that is commit-pending in H is either committed or aborted in H'. Clearly, every history in a set $\mathit{Complete}(H)$ is complete.

For example, consider the following history H_3:

$$H_3 \quad = \quad \langle \mathit{write}_1(x,1), \mathit{tryC}_1, \mathit{read}_2(x,1) \rangle.$$

Then, in each history in set $\mathit{Complete}(H_3)$: (1) transaction T_1 is either committed or aborted, and (2) transaction T_2 is (forcefully) aborted. The following histories are some of the elements of $\mathit{Complete}(H_3)$:

$$\begin{aligned} H_3' &= \langle \mathit{write}_1(x,1), \mathit{tryC}_1, C_2, \mathit{read}_2(x,1), \mathit{tryC}_2, A_2 \rangle, \\ H_3'' &= \langle \mathit{write}_1(x,1), \mathit{tryC}_1, \mathit{read}_2(x,1), \mathit{tryC}_2, A_2, C_1 \rangle. \end{aligned}$$

Sequential specification of a shared object. We use the concept of a *sequential specification* to describe the semantics of shared objects, as in [30, 16]. Intuitively, a sequential specification of a shared object ob lists all sequences of operation executions on ob that are considered correct when executed outside any transactional context, e.g., in a standard, single-threaded application.[9] For example, the sequential specification of a register x, denoted by $\mathit{Seq}(x)$, is the set of all sequences of *read* and *write* operation executions on x, such that in each sequence that belongs to $\mathit{Seq}(x)$, every *read* (operation execution) returns the value given as an argument to the latest preceding *write* (regardless transaction identifiers). (In fact, $\mathit{Seq}(x)$ also contains sequences that end with a pending invocation of *read* or *write*, but this is a minor detail.) Such a set defines precisely the semantics of a read-write register in a single-threaded, non-transactional system.

More formally, let an *object-local history* of a shared object ob be any prefix S of a sequence of operation executions, such that $S|ob = S$. Then, a sequential specification $\mathit{Seq}(ob)$ of a shared object ob may be any prefix-closed set of object-local histories of that object. (A set Q of sequences is *prefix-closed* if, whenever a sequence S is in Q, every prefix of S is also in Q.)

Legal histories and transactions. Let S be any sequential history, such that every transaction in S, except possibly the last one, is committed. Intuitively, we will say that S is *legal* if S respects the sequential specifications of all the shared objects, operations on which are performed in S. Note that the meaning of the word "respects" is clear here, because in S no two transactions are concurrent and no transaction comes after a live or aborted transaction. More formally, a sequential history S is *legal* if, for every shared object $ob \in \mathit{Obj}$, subsequence $S|ob$ is in set $\mathit{Seq}(ob)$.

Let S be any complete sequential history. In general, for such a history the definition of a legal history does not necessarily apply, because there may be many aborted transactions in S. Thus, we will instead consider each transaction T_i in S separately (T_i being committed or aborted), together with all the committed transactions preceding T_i in S, and determine legality of so-constructed sequential history. More precisely, we say that a transaction $T_i \in S$ is *legal in* S, if the largest subsequence S' of S, such that, for every transaction $T_k \in S'$, either (1) $k = i$, or (2) T_k is committed and $T_k \prec_S T_i$, history S' is legal.

5. Opacity

Opacity is a safety property that captures the intuitive requirements that (1) all operations performed by every *committed* transaction appear as if they happened at some single, indivisible point during the transaction lifetime, (2) no operation performed by any *aborted* transaction is ever visible to other transactions (including live ones), and (3) every transaction always observes a *consistent* state of the system.

5.1 Intuition

The first requirement above is captured by the classical notion of global atomicity [30]. This notion stipulates that after removing all non-committed transactions from any history H, the resulting history H' is equivalent to some sequential history S that respects the sequential specification of every shared object (i.e., is legal). Additionally, we also require that S preserves the real-time ordering of transactions in H'.

Global atomicity (even if combined with recoverability), however, does not guarantee the other two requirements, as explained in Section 3. Intuitively, when a transaction T_i accesses some shared object, T_i should observe the changes done to the shared object by all transactions that committed before T_i started, but should not see any modifications done by transactions that are still live (and not commit-pending) or aborted. Moreover, no transaction should observe the changes done by T_i until T_i *commits*, i.e., until some unique point in time, between commit-try and commit events of

[9] An operation execution specifies a transaction identifier, but the identifier can be treated as a part of the arguments of the executed operation. In fact, in most cases, the semantics of an operation does not depend on the transaction that issues this operation.

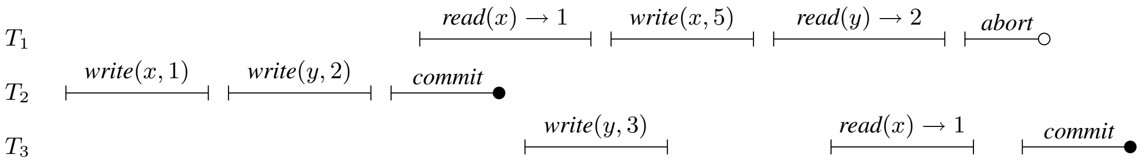

Figure 2. An opaque history H_5

T_i, at which all the changes done by T_i become instantaneously visible.

To see how we capture the second and third requirement, consider complete histories only. The key idea is to check, for every such history H, that every (aborted or committed) transaction T_k in H observes a state of the system produced by a sequence of all committed transactions preceding T_k, and some committed transactions concurrent with T_k. More precisely, we require that there exists a sequential history S, such that (1) S is equivalent to H, (2) S preserves the real-time order of H, and (3) every transaction in S is legal in S. The requirement (3) means that, for every transaction T_k in S, the longest subsequence of S made of (1) all committed transactions preceding T_k in S, and (2) transaction T_k itself, is a legal history, i.e., a history that respects the semantics of all operations on shared objects. In a sense, S corresponds to the (total) order in which transactions appeared to happen (instantaneously) in history H. As we already mentioned, legality is trivial to determine for complete sequential histories, in which no transaction (except possibly the last one) is aborted, given the semantics (i.e., the sequential specifications) of all shared objects accessed by transactions in S.

As for an incomplete history H, we transform it into a complete history H' by committing or aborting every live transaction in H. A transaction that is live and *not* commit-pending in H can only be aborted in H': before a transaction T_k invokes a commit-try event, the semantics of T_k is the same as of an aborted transaction, i.e, no changes made by T_k to shared objects should be visible to other transactions. A transaction that is commit-pending in H can be either aborted or committed in H': all the changes made by a transaction to shared objects become visible at some single unique point in time between commit-try and commit events of the transaction.

5.2 Definition

DEFINITION 1. *A history H is* opaque *if there exists a sequential history S equivalent to some history in set $\mathit{Complete}(H)$, such that (1) S preserves the real-time order of H, and (2) every transaction $T_i \in S$ is legal in S.*

Two points of the definition contain subtleties that need further explanation. Firstly, the step of transforming a given history H into a complete history results in a set of histories $\mathit{Complete}(H)$. The reason why this set may contain many elements is the dual semantic of commit-pending transactions that may be considered as either committed or aborted. Basically, the exact point in time at which a commit-pending transaction T_i begins to appear as committed to other transactions is not visible to a user, and thus not expressed as an event in a history. While in many TM implementations there is a single instruction at which a commit-pending transaction commits, the safety guarantees that a TM provides to a user should be expressed only with the events that the user can observe. Thus, in a sense, a TM should be treated as a black box the properties of which are defined using its external interface.

There is, however, a subtlety in the way we treat commit-pending transactions. Basically, if a transaction is commit-pending, its changes to shared objects may be already visible to some transactions and, at the same time, not yet visible to other ones. For example, consider the following history H_4 (x and y are registers with initial value of 0):

$$\begin{aligned} H_4 \;\; = \;\; & \langle \mathit{read}_1(x, 0), \mathit{write}_2(x, 5), \mathit{write}_2(y, 5), \mathit{tryC}_2, \\ & \mathit{read}_3(y, 5), \mathit{read}_1(y, 0) \rangle. \end{aligned}$$

In H_4, transaction T_1 appears to happen before T_2, because T_1 reads the initial values of registers x and y that are modified by T_2. Transaction T_3, on the other hand, appears to happen after T_2, because it reads the value of y written by T_2. Because the three transactions in H_4 are pairwise concurrent, sequential history $S = H_4|T_1 \cdot \langle \mathit{tryC}_1, A_1 \rangle \cdot H_4|T_2 \cdot \langle C_2 \rangle \cdot H_4|T_3 \cdot \langle \mathit{tryC}_3, A_3 \rangle$, equivalent to some history in $\mathit{Complete}(H_4)$, trivially preserves the real-time order of H_4. Because every transaction is legal in S, history H_4 is opaque. However, at first, it may seem wrong that the *read* operation of transaction T_3 returns the value written to y by the commit-pending transaction T_2 while the following *read* operation, by transaction T_1, returns the old value of y. But if T_1 read value 5 from y, then opacity would be violated, because T_1 would observe an inconsistent state of the system ($x = 0$ and $y = 5$). Thus, letting T_1 read 0 from y is the only way to prevent T_1 from being forcefully aborted without violating opacity. Multi-version TMs, like JVSTM and LSA-STM, indeed use such optimizations to allow long read-only transactions to commit despite concurrent updates performed by other transactions. In general, it seems that forcing the order between operation executions of different transactions to be preserved, in addition to the real-time order of transactions themselves, would be too strong a requirement.

The second subtlety in the definition of opacity is the fact that it does not require every prefix of an opaque history to be also opaque. Thus, the set of all opaque histories is not prefix-closed. However, a history of a TM is generated progressively and at each time the history of all events issued so far must be opaque. Hence, there is no need to enforce prefix-closeness in the definition of opacity, which should be as simple as possible.

5.3 Example

To illustrate our definition, consider the following history H_5, of three transactions accessing two registers (x and y), corresponding to the execution depicted in Figure 2:

$$\begin{aligned} H_5 \;\; = \;\; & \langle \mathit{write}_2(x, 1), \mathit{write}_2(y, 2), \mathit{tryC}_2, \\ & \mathit{inv}_1(x, \mathit{read}, \bot), \\ & C_2, \\ & \mathit{inv}_3(y, \mathit{write}, 3), \\ & \mathit{ret}_1(x, \mathit{read}, 1), \mathit{inv}_1(x, \mathit{write}, 5), \\ & \mathit{ret}_3(y, \mathit{write}, \mathit{ok}), \\ & \mathit{ret}_1(x, \mathit{write}, \mathit{ok}), \mathit{inv}_1(y, \mathit{read}, \bot), \\ & \mathit{inv}_3(x, \mathit{read}, \bot), \\ & \mathit{ret}_1(y, \mathit{read}, 2), \mathit{tryC}_1, \\ & \mathit{ret}_3(x, \mathit{read}, 1), \mathit{tryC}_3, \\ & A_1, \\ & C_3 \rangle. \end{aligned}$$

Clearly, $\mathit{Complete}(H_5) = \{H_5\}$ and $\prec_{H_5} = \{(T_2, T_3)\}$: there is no live transaction in H_5 and T_1 is concurrent with T_2 and T_3

in H_5. Therefore, we can find three sequential histories that are equivalent to H_5 and preserve the relation $\prec_{H_5}$ (thus satisfying real-time order). However, T_1 reads from x the value that has been written by committed transaction T_2. Thus, a sequential history in which T_1 precedes T_2 is not legal. Similarly, T_3 cannot precede T_1: T_1 reads from y the value written by T_2 and not the value written by the committed transaction T_3. Consider the following sequential history $S = H_5|T_2 \cdot H_5|T_1 \cdot H_5|T_3$. Clearly, S is equivalent to H_5 and preserves the real-time order of H_5. Furthermore, every transaction is legal in S, because sequential histories $H_5|T_2$, $H_5|T_2 \cdot H_5|T_1$, and $H_5|T_2 \cdot H_5|T_3$ are legal. Therefore, history H_5 is opaque.

However, complete history H_1 depicted in Figure 1 is not opaque for the following reason. Consider any sequential history S equivalent to $H_1 \in \mathit{Complete}(H_1) = \{H_1\}$. Because $T_1 \prec_{H_1} T_2$ and $T_1 \prec_{H_1} T_3$, history S may only be one of the following: (1) $H_1|T_1 \cdot H_1|T_2 \cdot H_1|T_3$, or (2) $H_1|T_1 \cdot H_1|T_3 \cdot H_1|T_2$. However, in both cases transaction T_2 is not legal in S. That is because: (1) in the first case, the second read of T_2 returns 2 instead of 0 (assuming the initial value of y is 0), and (2) in the second case, the first read of T_2 returns 1 instead of 2 (the value written by T_3).

5.4 Graph Characterization

Representing transactions as graph nodes and the causal relation between them as edges helps visualize a given history. Expressing opacity in terms of the acyclicity of such a graph, on the other hand, makes it easier to prove that the corresponding history is, or is not, opaque (we use this in proving our complexity lower bound). In this section, we present a framework, inspired by the works on 1-copy serializability [2], that allows for such a graph-based interpretation of opacity.

We focus here on histories in which every shared object used by a transaction is a read-write register. To simplify the discussion (but without loss in generality), we assume that (1) no two write operations write the same value to the same object (say, some local timestamp and a unique writer's id is added to the value), and (2) each history starts with an initializing, committed transaction T_0 that writes some values to every register.

Let H be a history and T_i be a transaction in H. A read operation (execution) $\mathit{read}_i(r, v) \in H|T_i$ is *local* if it is preceded in $H|T_i$ by a write operation $\mathit{write}_i(r, v')$. A write operation $\mathit{write}_i(r, v)$ is *local* if it is followed in $H|T_i$ by a write operation $\mathit{write}_i(r, v')$. A history H' is the *non-local subhistory* of H, denoted $\mathit{nonlocal}(H)$, if H' is the longest subsequence of H that does not contain any local operation execution.

We say that T_i *reads (value v from) register* r in H, if $H|T_i$ contains $\mathit{read}_i(r, v)$. We say that T_i *writes (value v to) register* r in H, if $H|T_i$ contains $\mathit{inv}_i(r, \mathit{write}, v)$. We say that a transaction T_k *reads (register r) from* transaction T_i, if T_i writes a value v to r and T_k reads value v from r.

A history H is *locally-consistent* if, for every transaction T_i and every local operation $\mathit{read}_i(r, v) \in H|T_i$, the latest write operation in $(H|T_i)|r$ that precedes $\mathit{read}_i(r, v)$ is $\mathit{write}_i(r, v)$. A history H is *consistent* if (1) H is locally-consistent, and (2) for every transaction $T_i \in H$, if T_i reads value v from register r in history $\mathit{nonlocal}(H)$, then some transaction T_k writes value v to r in $\mathit{nonlocal}(H)$.

Let H be a history, $\ll$—a total order on the set of transactions in H, and V—a subset of the set of commit-pending transactions in H. We call an *opacity graph* $\mathit{OPG}(H, \ll, V)$ a directed, labeled graph constructed as follows. Every transaction T_i in H corresponds to a vertex in $\mathit{OPG}(H, \ll, V)$, and the vertex is labelled L_{vis} if T_i is in set V or is committed, or L_{loc} otherwise. For every two transactions $T_i, T_k \in H$, there is an edge (T_i, T_k) in $\mathit{OPG}(H, \ll, V)$ in any of the following cases:

1. If $T_i \prec_H T_k$; then the edge is labelled L_{rt};
2. If T_k reads from T_i; then the edge is labelled L_{rf};
3. If $T_i \ll T_k$ and T_i reads some register r that is written by T_k; then the edge is labelled L_{rw};
4. If $T_i \in V$ or T_i is committed, and there exists a transaction T_m and a register r, such that $T_i \ll T_m$, T_i writes to r, and T_m reads r from T_k; then the edge is labelled L_{ww}.

We say that opacity graph $\mathit{OPG}(H, \ll, V)$ is *well-formed* if the following condition is satisfied: if T_i is a vertex of $\mathit{OPG}(H, \ll, V)$ labeled L_{loc}, then there is no edge (T_i, T_k) labelled L_{rf}, for any vertex T_k of $\mathit{OPG}(H, \ll, V)$.

The following theorem, proved in the extended version of this paper [10], establishes a formal relationship between the opacity of a given history H and the properties of the opacity graph of H.

THEOREM 2. *A history H is opaque if, and only if, (1) H is consistent, and (2) there exists a total order $\ll$ on the set of transactions in H and a subset V of the set of commit-pending transactions in H, such that $\mathit{OPG}(\mathit{nonlocal}(H), \ll, V)$ is well-formed and acyclic.*

6. A Complexity Lower Bound

A crucial choice in a TM implementation is that of visible vs. invisible read strategy [19]. To illustrate this, consider a situation in which a transaction T_i invokes a read-only operation op on a shared object ob. The TM implementation that executes T_i, at some process p_k, and receives the invocation event of op, must somehow get the current state of ob, apply op locally and return the resulting value to T_i. Additionally, p_k may also write somewhere in base (hardware) shared objects the information that T_i is currently accessing ob, in which case the operation op becomes *visible* to other processes. If p_k never modifies any base shared object when processing op, then the operation is always *invisible* to other processes.

A practical advantage of invisible reads is that p_k, while executing op, does not invalidate any processor cache lines. For read-dominated applications, the traffic on the bus between processors is thus greatly reduced, and so the overall throughput of operations is potentially larger. The problem, however, is that while T_i reads some shared objects, other transactions may at any time modify these objects, because read-only operations of T_i are visible only to p_k. An additional cost of per-operation *validation* might thus be required to guarantee that T_i always observes a consistent state.[10]

We make use of opacity to precisely determine when invisible reads indeed induce a high operation complexity. Basically, we prove a lower bound of $\Omega(k)$ (where $k = |\mathit{Obj}|$) on the worst-case operation complexity for every TM implementation that uses invisible reads, (1) is single-version, and (2) does never abort a transaction unless it conflicts with some other live transaction. If any of the two conditions is not required, or if we allow visible reads, one can devise a TM implementation with operation complexity not bounded by $\Omega(k)$. That is, the lower bound does not hold for TMs that use visible reads (e.g., RSTM), are multi-version (e.g., JVSTM), or provide strictly weaker progress guarantees (e.g., TL2).

Opacity is crucial here. One can devise an algorithm that ensures a combination of global atomicity (with real-time ordering) and strict recoverability instead of opacity, uses invisible reads and satisfies properties (1) and (2) above, and that has constant operation complexity (e.g., such algorithm is given in the extended version of this paper [10]). In this sense, our bound highlights

[10] The problem of visible vs. invisible reads is similar to the "readers must write" issue in register implementations [17].

the complexity gap between systems that support full isolation of transactional code from the outside environment, e.g., databases or virtual machines for languages that can provide "sandboxing" of code blocks, and those that do not. The former systems can render aborted transactions completely harmless and so a correctness criterion weaker than opacity can be used.

Before giving the outline of the proof, we define certain elements that underly the very notion of a TM implementation, and give an intuition behind the properties used in the proof. (For details of the proof, as well as precise definitions of the elements introduced here, refer to [10].)

6.1 TM Implementations

A *TM implementation* is an algorithm that interprets the events issued by transactions and generates matching responses. The algorithm is executed by a number of *processes* that communicate by issuing *instructions* on *base* shared objects. In a single *step*, a process issues a single instruction on a single base shared object. We consider that each transaction is executed by a single process, and that each process executes transactions *sequentially*.

Roughly speaking, we also assume that every TM implementation satisfies the following conditions: it does not require information about more than a constant number of shared objects to be retrieved from a single base shared object (i.e., in a single step), and it does not force a process to execute steps of the TM algorithm if this process does not have any pending invocation event (i.e., it does not use any specific background services). All TM implementations we know of satisfy these properties.

Intuitively, we say that a TM implementation *I*:

1. Is *progressive* if *I* forcefully aborts a transaction T_i only when there is a time t at which T_i *conflicts* with another, concurrent transaction T_k that is not committed or aborted by time t (i.e., T_k is live at t);[11] we say that two transactions conflict if they access some common shared object.[12]
2. Is *single-version* if *I* stores only the latest committed state of any given shared object in base shared objects (as opposed to multi-version TM implementations, e.g., [5, 25]).
3. Uses *invisible reads* if no base shared object is modified when a transaction performs a read-only operation on a shared object.

6.2 Complexity Result

Roughly speaking, the time complexity of a given TM implementation is the maximum possible number of steps that a process may execute while processing a single operation issued by a transaction, i.e., from the operation invocation event until the matching response event. We prove the following result.

THEOREM 3. *Every progressive, single-version TM implementation that ensures opacity and uses invisible reads has the time complexity of* $\Omega(k)$*, where* $k = |Obj|$.

Proof. (Intuition; the full proof is in [10]) Consider any progressive, single-version TM implementation that ensures opacity and uses invisible reads. Consider the following scenario: two transactions, T_1 and T_2, executed by two different processes, p_1 and p_2, respectively, are accessing only read/write objects. Transaction T_1 reads some $\Theta(k)$ objects. Then, T_2 writes some $\Theta(k)$ objects and commits. Now, if T_1 invokes a read operation on an object r that has been modified by T_2 (and that has not been read by T_1 so far), then T_1 will be returned the value written to r by T_2 (because the TM implementation is single-version). However, p_1 needs to determine whether any other object read by T_1 has been updated by T_2. If yes, T_1 has to be aborted (instead of returning from the read operation): otherwise opacity would be violated. Indeed, then T_1 would read some values before T_2 overwrote them with different ones, and some values written by T_2. If no, p_1 has to let T_1 eventually commit; this is because the TM implementation is progressive (and we assume that T_1 does not invoke *tryA*$_1$).

The key point is that because the TM implementation uses invisible reads, p_2 does not know which objects were read by T_1. Thus, p_2 cannot help p_1 detect a situation in which T_2 has updated an object that has just been read by T_1 before. Now, because only constant-size information can be obtained by p_1 in each step, p_1 needs to execute $\Omega(k)$ steps to be sure whether it has to abort T_1 immediately or let T_1 commit. □

Even from the intuition of the proof, it should be clear that all the properties we require, i.e., invisible reads, progressiveness, and the single-version scheme, as well as the assumptions we make, are necessary for the lower bound to hold. This is confirmed by the already mentioned counterexample TM implementations that have the time complexity either constant or at least independent of k (e.g., RSTM, JVSTM, TL2, etc.).

The lower bound is tight because DSTM and ASTM are progressive and single-version, ensure opacity and use invisible reads, and have the time complexity of $\Theta(k)$ (with most contention managers). It is worth noting that TL2 has a constant time complexity, although it ensures opacity, uses invisible reads, and is single-version. That is because TL2 is not progressive: it may forcefully abort a transaction T_i that conflicts with a concurrent transaction T_k, even if T_i invokes a conflicting operation after T_k commits.

7. Concluding Remarks

This paper presents opacity: a correctness criterion for TM systems. Opacity constitutes a first step towards a theory of transactional memory. Such a theory is badly missing to reason about the correctness of TM algorithms and establish underlying optimality results and inherent trade-offs, as well as serve as a reference point for weaker models that would be more efficient to implement (cf. serializability vs. lower isolation levels in databases). Many related issues were, however, not addressed in this paper.

In particular, we considered a concurrency scheme where all accesses to shared objects are performed within transactions, and we focused on a flat transaction model.

It is often argued that, in practice, transactions might be mixed with non-transactional code [3], especially when coping with legacy components. A model where transactions would observe concurrent updates made by non-transactional code, and where changes made by live transactions would be visible to operations outside transactions is, clearly, imprecise. It is preferable to require that every non-transactional operation has the semantics of a single transaction. This preserves the illusion that transactions appear as if they were executed instantaneously and disallows race conditions between transactional and non-transactional code. We can encompass such a model in our context by encapsulating every non-transactional operation into a *committed* transaction.[13] Clearly, an actual transactional memory implementation may take advantage of the fact that such a transaction contains only a single operation and can thus be executed more efficiently (e.g., without logging changes).

The model within which we express the notion of opacity can also be extended to account for nested transactions (with either

[11] The property resembles the concept of C-respecting in [27].

[12] For simplicity, we do not distinguish between read-only and update accesses here.

[13] The ability to integrate transactional and non-transactional code would thus be expressed in our context in the form of a progress property stipulating that such single operation transactions are never forcefully aborted.

closed [21] or open [22] nesting semantics). Basically, we can treat events of each committed nested transaction as if they were executed directly by the parent transaction. Aborted and live nested transactions can be accounted for in a similar way as we deal with aborted and live (flat) transactions in the definition of opacity. The main difference here is that a nested transaction should observe the changes done by its parent transaction. We can capture this by always considering operations of a nested transaction together with all the preceding operations of its parent transaction.

Finally, it is also worthwhile noticing that opacity, by itself, does not say when transactions should commit. Our work is in this sense complementary to [9, 27] which define progress properties and classify contention management strategies. It would be interesting to see which combinations with opacity are possible and at what cost.

Acknowledgments

We would like to thank Hagit Attiya, Pascal Felber, Christof Fetzer, Seth Gilbert, Tim Harris, Eshcar Hilel, and Nir Shavit for interesting discussions on the topic of this paper, as well as the anonymous reviewers for their helpful comments.

References

[1] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD'95)*, pages 1–10, New York, NY, USA, 1995. ACM Press.

[2] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[3] C. Blundell, E. Lewis, and M. M. K. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2), 2006.

[4] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, 17(9):954–960, 1991.

[5] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. In *Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL); in conjunction with the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, 2005.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, 2006.

[7] R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

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

[9] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC'05)*, 2005.

[10] R. Guerraoui and M. Kapałka. Opacity: A correctness condition for transactional memory. Technical Report LPD-REPORT-2007-004, EPFL, May 2007. `http://lpd.epfl.ch/kapalka/files/opacity-techreport07.pdf`.

[11] V. Hadzilacos. A theory of reliability in database systems. *Journal of the ACM*, 35(1):121–145, 1988.

[12] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proceedings of ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI'06)*, 2006.

[13] M. Herlihy. SXM software transactional memory package for C#. `http://www.cs.brown.edu/~mph`.

[14] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22th Annual ACM Symposium on Principles of Distributed Computing (PODC'03)*, pages 92–101, 2003.

[15] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. May 1993.

[16] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, June 1990.

[17] L. Lamport. On interprocess communication–part I: Basic formalism, part II: Algorithms. *Distributed Computing*, 1(2):77–101, 1986.

[18] V. J. Maranthe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)*, pages 354–368, 2005.

[19] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, 2006.

[20] J. E. B. Moss. Nested transactions and reliable distributed computing. In *Second IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 33–39, 1982.

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

[22] J. E. B. Moss. Open nested transactions: Semantics and support. In *Poster presented at Workshop on Memory Performance Issues (WMPI'06)*, Feb. 2006.

[23] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the ACM SIGPLAN 2007 Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*, pages 68–78, Mar. 2007.

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

[25] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, 2006.

[26] T. Riegel, P. Felber, and C. Fetzer. Snapshot isolation for software transactional memory. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, 2006.

[27] M. L. Scott. Sequential specification of transactional memory semantics. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, 2006.

[28] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC'95)*, pages 204–213. Aug 1995.

[29] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, 2006.

[30] W. E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, April 1989.

[31] A. Y. Zomaya, editor. *Parallel and Distributed Computing Handbook*. McGraw-Hill, 1996.