Opacity: A Correctness Condition for Transactional Memory[*]

Rachid Guerraoui   Michał Kapałka

Distributed Programming Laboratory, School of Computer and Communication Sciences, EPFL

December 5, 2007

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.

[*]EPFL Technical Report LPD-REPORT-2007-004. Submitted for publication.

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 hand-crafted 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, 12, 18, 5, 19, 11, 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, 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 [12], 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, as we show in this paper, our bound does not hold for serializability, even when considered in its strict form to account for real-time order and combined with recoverability [10]. 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 also give a TM implementation that serves as an counterexample that the lower bound does not hold if

[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.

opacity is not required. 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].

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 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 runtime 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

[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.

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.

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 [10] 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.

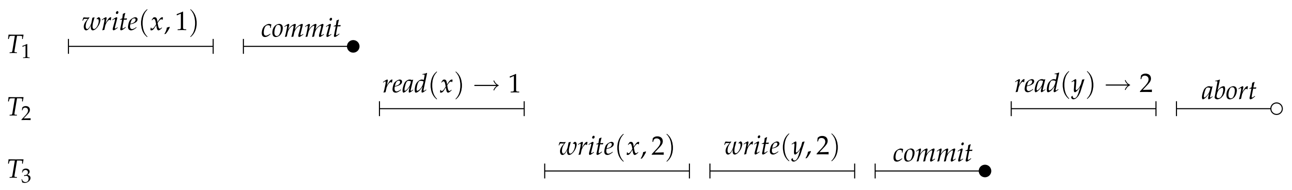

Figure 1: A history that satisfies global atomicity (with real-time ordering guarantees) and recoverability, but in which an aborted transaction (T_2) accesses an inconsistent state of the system (x and y are simple variables/objects that implement *read* and *write* operations)

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 8, 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.

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

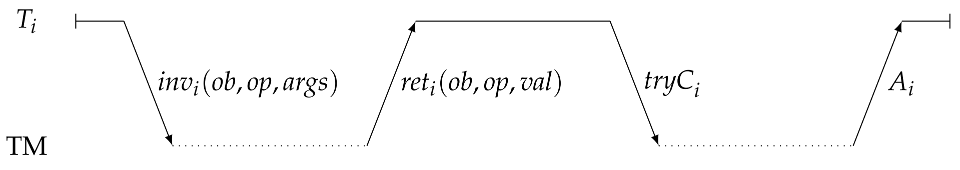

Figure 2: Events of a transaction T_i

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.

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 $write(v)$ 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 $Trans = \{T_1, T_2, \ldots\}$. 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 $inv_i(ob, op, args)$ and expects a matching *operation response* event $ret_i(ob, op, val)$, 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).

Figure 2 illustrates an example interaction of a transaction T_i with a TM. The transaction accesses only one shared object, ob, using a single operation op. Then, T_i issues a commit-try event that informs the TM that T_i wants to commit. However, the commit-try request of T_i is rejected and T_i gets aborted (receives an abort event).

An *operation execution* is a pair of an operation invocation event and a matching operation response event. That is, an operation execution $exec_i(ob, op, args, val)$ is a sequence $\langle inv_i(ob, op, args),\ ret_i(ob, op, val)\rangle$.[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 $read_i(r, v)$ a *read* operation execution on register r, by transaction T_i, returning value v, and by $write_i(r, v)$ a *write* operation execution on register r, by T_i, with value v given as an argument. More formally, $read_i(r, v) = exec_i(r, read, \perp, v)$, and $write_i(r, v) = exec_i(r, write, v, ok)$.

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]

[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.

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 multi-processor 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) $\langle \mathit{inv}_i(ob, op, args), A_i \rangle$ (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:

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

or, using more verbose notation:

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

Clearly, there is no pending invocation event in H_1. The histories $H_1|T_2$ and $H_1|x$ are as follows:

$$\begin{aligned} H_1|T_2 &= \langle \mathit{read}_2(x,1), \mathit{read}_2(y,2), \mathit{tryC}_2, A_2 \rangle, \\ H_1|x &= \langle \mathit{write}_1(x,1), \mathit{read}_2(x,1), \mathit{write}_3(x,2) \rangle. \end{aligned}$$

The following history H_2 is one of the histories that are equivalent to H_1:

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

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*.

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. In the following prefix H_1' of H_1, transaction T_1 is live:

$$H_1' = \langle \mathit{write}_1(x,1) \rangle,$$

while in the following prefix H_1'' of H_1, transaction T_1 is commit-pending:

$$H_1'' = \langle \mathit{write}_1(x,1), \mathit{tryC}_1 \rangle.$$

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 $T_i \nprec_H T_j$ and $T_j \nprec_H T_i$.

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:

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

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.

[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.

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, let $\mathit{visible}_S(T_i)$ denote 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 $T_k \prec_S T_i$. Then, we say that a transaction $T_i \in S$ is *legal in S*, if history $\mathit{visible}_S(T_i)$ is legal.

For example, consider the sequential history H_2 introduced before. Then:

$$\begin{aligned} \mathit{visible}_{H_2}(T_1) &= \langle \mathit{write}_1(x,1), \mathit{tryC}_1, C_1 \rangle, \\ \mathit{visible}_{H_2}(T_3) &= \mathit{visible}_{H_2}(T_1) \cdot \\ &\qquad \langle \mathit{write}_3(x,2), \mathit{write}_3(y,2), \mathit{tryC}_3, C_3 \rangle, \\ \mathit{visible}_{H_2}(T_2) &= H_2. \end{aligned}$$

Histories $\mathit{visible}_{H_2}(T_1)$ and $\mathit{visible}_{H_2}(T_3)$ are both legal, because:

$$\begin{aligned} \mathit{visible}_{H_2}(T_1)|x = \langle \mathit{write}_1(x,1) \rangle &\in \mathit{Seq}(x), \\ \mathit{visible}_{H_2}(T_1)|y = \langle\, \rangle &\in \mathit{Seq}(y), \\ \mathit{visible}_{H_2}(T_3)|x = \langle \mathit{write}_1(x,1), \mathit{write}_3(x,2) \rangle &\in \mathit{Seq}(x), \\ \mathit{visible}_{H_2}(T_3)|y = \langle \mathit{write}_3(y,2) \rangle &\in \mathit{Seq}(y). \end{aligned}$$

Hence, transactions T_1 and T_3 are legal in H_2. However, history $\mathit{visible}_{H_2}(T_2)$ is not legal, because the following sequence violates the sequential specification of a register (i.e., it is not in set $\mathit{Seq}(x)$):

$$\begin{aligned} \mathit{visible}_{H_2}(T_2)|x = \langle &\mathit{write}_1(x,1), \\ &\mathit{write}_3(x,2), \mathit{read}_2(x,1) \rangle. \end{aligned}$$

Therefore, transaction T_2 is not legal in H_2.

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 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 $Complete(H)$, 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 $Complete(H)$. 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):

$$\begin{aligned} H_4 \quad = \quad & \langle read_1(x,0), write_2(x,5), write_2(y,5), tryC_2, \\ & read_3(y,5), read_1(y,0) \rangle. \end{aligned}$$

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 $S = H_4|T_1 \cdot \langle tryC_1, A_1 \rangle \cdot H_4|T_2 \cdot \langle C_2 \rangle \cdot H_4|T_3 \cdot \langle tryC_3, A_3 \rangle$, equivalent to some history in $Complete(H_4)$, 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.

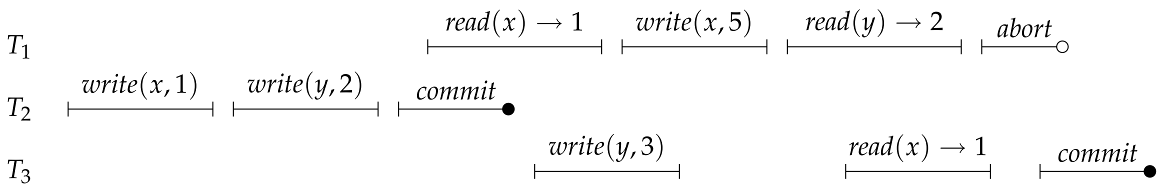

Figure 3: An opaque history H_5

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 3:

$$\begin{aligned} H_5 \;=\; & \langle write_2(x,1), write_2(y,2), tryC_2, \\ & inv_1(x, read, \bot), \\ & C_2, \\ & inv_3(y, write, 3), \\ & ret_1(x, read, 1), inv_1(x, write, 5), \\ & ret_3(y, write, ok), \\ & ret_1(x, write, ok), inv_1(y, read, \bot), \\ & inv_3(x, read, \bot), \\ & ret_1(y, read, 2), tryC_1, \\ & ret_3(x, read, 1), tryC_3, \\ & A_1, \\ & C_3\rangle. \end{aligned}$$

Clearly, $Complete(H_5) = \{H_5\}$ 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 $H_1 \in Complete(H_1) = \{H_1\}$. 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).

6 A Graph Characterization of Opacity

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) $read_i(r, v) \in H|T_i$ is *local* if it is preceded in $H|T_i$ by a write operation $write_i(r, v')$. A write operation $write_i(r, v)$ is *local* if it is followed in $H|T_i$ by a write operation $write_i(r, v')$. A history H' is the *non-local subhistory* of H, denoted $nonlocal(H)$, 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 $read_i(r, v)$. We say that T_i *writes (value v to) register r* in H, if $H|T_i$ contains $inv_i(r, write, v)$. 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.

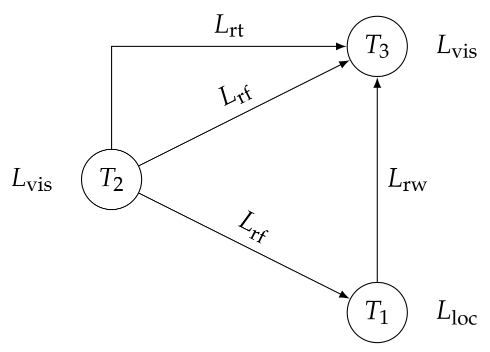

Figure 4: Opacity graph $OPG(H_5, \ll, \emptyset)$

A history H is *locally-consistent* if, for every transaction T_i and every local operation $read_i(r, v) \in H|T_i$, the latest write operation in $(H|T_i)|r$ that precedes $read_i(r, v)$ is $write_i(r, v)$. 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 $nonlocal(H)$, then some transaction T_k writes value v to r in $nonlocal(H)$.

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* $OPG(H, \ll, V)$ a directed, labeled graph constructed as follows. Every transaction T_i in H corresponds to a vertex in $OPG(H, \ll, V)$, 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) (denoted $T_i \longrightarrow T_k$) in $OPG(H, \ll, V)$ in any of the following cases:

1. If $T_i \prec_H T_k$; then the edge is labelled L_{rt} (and denoted $T_i \xrightarrow{\text{rt}} T_k$);
2. If T_k reads from T_i; then the edge is labelled L_{rf} (and denoted $T_i \xrightarrow{\text{rf}} T_k$);
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} (and denoted $T_i \xrightarrow{\text{rw}} T_k$);
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} (and denoted $T_i \xrightarrow{\text{ww}} T_k$).

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

For example, consider history H_5 introduced in Section 5.3 and depicted in Figure 3. Clearly, there is no local read or write operation in H_5, and so $nonlocal(H_5) = H_5$. Also, there is no pending operation invocation event in H_5. Let $\ll$ be the following total order of transactions in H_5:

$$T_2 \ll T_1 \ll T_3.$$

Figure 4 depicts the opacity graph $OPG(H_5, \ll, \emptyset)$ of history H_5. As the graph is well-formed and acyclic, and as H_5 is consistent, we have shown that H_5 is opaque.

The following theorem, 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 $OPG(nonlocal(H), \ll, V)$ is well-formed and acyclic.*

Proof. ($\Rightarrow$) Let H be an opaque history. Clearly, H must be consistent. By opacity, there exists a sequential history S equivalent to some history in set $Complete(H)$, such that S preserves the real-time order of H and every transaction $T_i \in S$ is legal in S. Let V be the set of transactions that are committed in S, and $\ll$ be the relation $\prec_S$. We will show in the following that graph $OPG(nonlocal(H), \ll, V)$ is well-formed and acyclic.

Let us denote history $nonlocal(H)$ by H_{nl}, and graph $OPG(H_{\text{nl}}, \ll, V)$ by G. Assume, by contradiction, that G is not well-formed. This means that there are two transactions $T_i, T_k \in H$, such that T_k reads some register r from T_i in H_{nl}, and T_i is either live, but not in V, or aborted. If $T_k \prec_S T_i$, then transaction T_k could not be legal in S, because T_k reads from r the value that is written (only) by T_i, and T_i comes after T_k in S. Therefore, it must be that $T_i \prec_S T_k$. Let S_k be the history $visible_S(T_k)$. By our definition of set V, transaction T_i must be aborted in S, and so $T_i \notin S_k$. Thus, history S_k is not legal, because T_k reads in S_k from register r a value that is never written in S_k. Hence, T_k is not legal in S—a contradiction with the assumption that every transaction in S is legal in S.

Assume now, by contradiction, that graph G is not acyclic. Then, because relation $\ll$ is a total order, there must be some transactions T_i and T_k, such that $T_i \ll T_k$ and there is an edge $T_k \longrightarrow T_i$ in G. Clearly, it is impossible that $T_k \xrightarrow{\text{rw}} T_i$ because $T_i \ll T_k$. It also cannot be that $T_k \xrightarrow{\text{rt}} T_i$, because if $T_k \xrightarrow{\text{rt}} T_i$, then $T_k \prec_H T_i$, and so, as $\prec_H = \prec_{H_{\text{nl}}} \subseteq \prec_S = \ll$, we would have that $T_k \ll T_i$, contradicting our assumption that $T_i \ll T_k$.

Assume that $T_k \xrightarrow{\text{rf}} T_i$. This means that there is a register r, such that T_k writes a value v to r and T_i reads v from r. Because $T_i \prec_S T_k$, history $S_i = visible_S(T_i)$ does not contain transaction T_k. Thus, S_i is not legal, because T_i reads value v from register r in S_i and no transaction writes value v to r in S_i. Hence, T_i is not legal in S—a contradiction with the assumption that every transaction in S is legal in S.

Therefore, graph G contains edge $T_k \xrightarrow{\text{ww}} T_i$. This means that $T_k \in V$ or T_k is committed in H, and there exists a transaction $T_m \in H$ and a register r, such that $T_k \ll T_m$, T_k writes to r and T_m reads r from T_i. But then T_k must be committed in S, and, by our assumptions, $T_i \ll T_k \ll T_m$. Thus, both (committed) transactions T_i

and T_k are in the sequential history $S_m = visible_S()$. Because, in S_m, transaction T_k writes to r after T_i writes to r (and commits), and because T_m reads r after T_k commits, T_m cannot read r from T_i. Hence, T_m is not legal in S—a contradiction with the assumption that every transaction in S is legal in S.

($\Leftarrow$) Let H be a consistent history, such that there exists a total order $\ll$ in the set of transactions in H and a subset V of the set of commit-pending transactions in H, such that graph $G = OPG(nonlocal(H), \ll, V)$ is well-formed and acyclic. We will show that H must be opaque.

Let sequence $T_{s_1}, \ldots, T_{s_m}$ be a result of a topological sorting of graph G. Let H_c be a history in set $Complete(H)$, such that every transaction T_i that is live in H is (1) committed in H_c if $T_i \in V$, or (2) aborted in H_c if $T_i \notin V$. Consider the sequential history $S = \langle H_c|T_{s_1} \cdot \ldots \cdot H_c|T_{s_m} \rangle$. Clearly, S is equivalent to H_c, and S preserves the real-time order of H (because of the $\xrightarrow{\text{rt}}$ edges in G).

Assume, by contradiction, that there is a transaction $T_i \in S$ that is not legal in S, i.e., for which the history $S_i = visible_S(T_i)$ is not legal. For simplicity, assume that T_i is the earliest transaction that is not legal in S, i.e., that every transaction preceding T_i in S is legal. This means that there exists a read operation $read_i(r, v)$ in S_i and either (1) there is no write operation on register r that precedes $read_i(r, v)$ in S_i, or (2) the latest write operation on r preceding $read_i(r, v)$ in S_i is $write_k(r, v')$, where $v' \neq v$. Clearly, we can exclude situation (1) because we assume that history H begins with a committed transaction T_0 that writes some initial value to every register, and that precedes every other transaction in H (and thus also in S and S_i, because of the $\xrightarrow{\text{rt}}$ edges in G). Hence, we assume situation (2).

Operation $read_i(r, v)$ cannot be local in S, because otherwise history H would not be locally-consistent. Also, because history H is consistent, there must be an event $inv_m(r, write, v)$ in H (issued by a transaction T_m). It must be that $m \neq i$ (i.e., T_m is different than T_i), as otherwise G would contain a cycle $T_i \xrightarrow{\text{rf}} T_i$. Hence, history S (and H) contains: (1) operations $write_k(r, v')$ and $read_i(r, v)$ (that also belong to S_i), where $k \neq i$ and $v \neq v'$, and (2) invocation event $e_m = inv_m(r, write, v)$, where $m \neq i$. This means that graph G contains edge $T_m \xrightarrow{\text{rf}} T_i$.

Assume first that $e_m \notin S_i$. This means that either transaction T_m is aborted in S, or $T_i \prec_S T_m$. If T_m is aborted in S, then T_m is not committed in H and not in set V, and so H is not consistent. Hence, $T_i \prec_S T_m$. But this is impossible, since G contains edge $T_m \xrightarrow{\text{rf}} T_i$ and relation $\prec_S$ must preserve the direction of edges in G (by the properties of a topological sort). Therefore, invocation event e_m must be in history S_i.

As T_i is the last transaction in history S_i, and because $m \neq i$, event e_m must precede operation $read_i(r, v)$. Also, e_m cannot be pending in S_i, i.e., S_i must contain operation execution $write_m(r, v)$. As $write_k(r, v')$ is the latest *write* preceding $read_i(r, v)$ in S_i, operation $write_m(r, v)$ must precede $write_k(r, v')$ in S_i. Hence, history $S_i|r$ must contain a sequence:

$$\langle write_m(r, v), \ldots, write_k(r, v'), read_i(r, v) \rangle.$$

Note also that T_m and T_k must be different transactions, as otherwise $write_m(r, v)$ would be a local operation, and so history H would not be consistent.

Transaction T_k is in S_i, and so T_k must be committed in S. Thus, T_k is either committed in H, or commit-pending in H and in set V. It must be that $T_k \ll T_i$, because otherwise there would be an edge $T_i \xrightarrow{\text{rw}} T_k$, and so it could not be that $T_k \prec_S T_i$ (by the properties of a topological sort). Therefore, there is an edge $T_k \xrightarrow{\text{ww}} T_m$—a contradiction with the assumption that T_m precedes T_k in the topological sort of G. $\square$

7 Low-Level Histories of TM Implementations

A *TM implementation* is an algorithm that interprets the events issued by transactions and sends back appropriate responses. Let $\Pi = \{p_1, \ldots, p_n\}$ be the set of *processes* that execute such an algorithm. We assume that each transaction is executed by a single process, and that each process executes transactions *sequentially*. We also assume that the events issued by transactions are not known in advance to processes.

Each process communicates with other processes by issuing *instructions* on *base shared objects*. We assume that instructions are atomic (i.e., linearizable [16]) and wait-free [13]. Therefore, we will only consider a single event for each instruction execution, called a *step*.[10] Each process might also perform local computations on objects inaccessible to other processes. Unless explicitly stated otherwise, we will allow processes to fail by *crashing*.[11] A process that crashes does not execute any further steps and does not receive or issue any further events.

A *low-level history* is a sequence of steps and events. If E is a low-level history, then $E|H$ denotes the longest subsequence of E that does not contain any step. We will say that $H = E|H$ is a *high-level* history *corresponding* to E. For every process p_i, we denote by $E|p_i$ the longest subsequence of E that contains only events and steps executed by process p_i. We assume that every low-level history E is *well-formed*. That is, history $E|H$ is well-formed and for every transaction $T_k \in E|H$, all events of T_k are in $(E|H)|p_i$ for some process p_i.

We say that a low-level history E is *opaque*, if $E|H$ is opaque. We say that a TM implementation is opaque if all its low-level histories are opaque.

We assume that every TM implementation guarantees the following: every transaction T_k which issues an invocation event eventually gets a response, unless the process executing T_k has crashed. More precisely, we assume

[10]Instead of a pair of invocation and matching response.

[11]Note that the proof of Theorem 7 does not rely on the assumption that processes can crash.

that, for every low-level history E and every process p_i, if $E|p_i$ is infinite, then there is no pending invocation event in $E|p_i$.

Consider a low-level history E and a process p_i. We say that E is *indistinguishable* for p_i from a low-level history E', if $E|p_i = E'|p_i$. If e is an operation execution in $(E|H)|p_i$, then every step in $E|p_i$ that is between the two events of e is said to be *corresponding* to e. If s is a step in $E|p_i$ and T_k is a transaction in $(E|H)|p_i$, then we say that s is *corresponding* to T_k if the latest event that precedes s in $E|p_i$ is an invocation event issued by T_k.

8 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.[12]

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 $k = |Obj|$) 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. As we show in this paper, 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. In this sense, our bound highlights 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 definitions of the properties used in the proof.

8.1 Properties of TM Implementations

Basic assumptions. Intuitively, we will assume that every TM implementation I satisfies the following conditions: (1) 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 (2) it does not force processes to execute steps of the TM algorithm when they do not have any pending invocation event (i.e., it does not use any specific background services).

More precisely, let Q be a subset of *Obj*, and c be a one-to-one function $c : Q \rightarrow Q$. For any two histories H and H', we will write $H \sim_c H'$ if H' is created from H by substituting every invocation event $inv_i(ob, op, args)$ with event $inv_i(c(ob), op, args)$, and every response event $ret_i(ob, op, val)$ with event $ret_i(c(ob), op, val)$ (for every process p_i, shared object $ob \in Q$, operation *op* and any values of *args* and *val*). For any two low-level histories E and E', we will write $E \sim_c E'$ if $E|H \sim_c E'|H$.

Let E be a low-level history and T_k be a transaction in $E|p_i$, for some process p_i. We say that T_k is *invisible* in E, if every low-level history $E' = E \cdot E_s$, where $E_s|p_i = \langle\,\rangle$, is indistinguishable for every process except p_i from history $E'' = E_r \cdot E_s$, where E_r is obtained from E by removing every step corresponding to T_k.

Then, for every low-level history E of I the following conditions are satisfied:

1. *Limited capacity.* For every process p_i, if $E = E_p \cdot E_m \cdot E_i$, where:
   - there is no live transaction in $E_p|p_k$ for every process p_k other than p_i, and the live transaction in $E_p|p_i$ (if any) is invisible in E_p,
   - $E_m|p_i = \langle\,\rangle$, and
   - E_i consists only of s steps of p_i,

   then there exists a set Q of size $|Obj| - O(s)$, such that for every one-to-one function $c : Q \rightarrow Q$ and every low-level history $E'_m \sim_c E_m$, if $E_p \cdot E'_m$ is a valid low-level history of I, then $E_p \cdot E'_m \cdot E_i$ is also a valid low-level history of I.

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

2. *No background threads.* Every step of E is corresponding to some transaction in $E|H$.

Properties of TM implementations. By conflict we mean, roughly speaking, a situation in which a number of concurrent transactions try to perform some operations on a common shared object. Intuitively, a TM implementation is progressive if it does not forcefully abort a transaction T_i unless T_i, at some point in time, has a *live* conflicting transaction.[13]

More precisely, let $Obj_H(T_i)$ denote the set of shared objects accessed by transaction T_i in history H, i.e., the set of shared objects, such that $ob \in Obj_H(T_i)$ if there is an event in $H|ob$ that is also in $H|T_i$. A *conflict* of a transaction T_k in a history H is each operation invocation event on a shared object in set $Obj_H(T_k)$, issued by a transaction concurrent to T_k and different from T_k (called a *conflicting transaction* of T_k).[14] Then we say that:

Definition 3 *A transactional memory I is* progressive *if, for every history H of I and every transaction $T_i \in H$ that is forcefully aborted, there exists a prefix H' of H and a transaction $T_k \in H'$ that is live in H', such that T_k is a conflicting transaction of T_i in H'.*

Intuitively, a TM implementation is single-version if, whenever a transaction T_i invokes an operation on a shared object *ob* for the first time, T_i accesses the latest committed state of *ob* (as opposed to multi-version TM implementations, e.g., [5, 25]). More precisely:

Definition 4 *A transactional memory I is* single-version *if every history H of I satisfies the following condition: for every prefix H' of H, every transaction T_i that is live in H', and every shared object ob, the longest subsequence H'' of H', such that $H''|T_i = (H''|T_i)|ob$, is a valid history of I.*

Roughly speaking, we say that an operation *op* of a shared object *ob* is *read-only* if *op* never modifies the state of object *ob*. Intuitively, a TM implementation uses *invisible reads* if no base shared objects are modified when a transaction performs a read-only operation on a shared object.

More precisely, operation *op* is read-only if, for every object-local history $S = S_1 \cdot S_2 \in Seq(ob)$ and every value of *args*, there exists a value *val*, such that $S_1 \cdot exec_i(ob, op, args, val) \cdot S_2 \in Seq(ob)$. We say that a transaction T_i is *read-only* if T_i invokes only read-only operations. Then we say that:

Definition 5 *A transactional memory I uses* invisible reads *if, for every low-level history E of I and every process p_i, E is indistinguishable for p_i from a low-level history E' that is obtained from E by removing every read-only operation execution e that is not in $(E|H)|p_i$ and every step corresponding to e.*

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

[14]This definition does not account for the fact that some operations do not really conflict, e.g., read-only ones. Clearly, an actual TM implementation may treat read-only or commutative operations in a special way.

Note that even if invisible reads are used, read-only operations of transactions executed by a process p_i are visible to p_i and thus may change the state of p_i.

Roughly speaking, the time complexity of a given TM implementation is the maximum possible number of steps that a process may execute while processing an operation issued by a transaction (i.e., from an operation invocation event until the matching response event). More precisely:

Definition 6 *The* time complexity *of a transactional memory I is the maximum number of steps corresponding to any operation execution, in any low-level history of I.*

8.2 Complexity Result

Theorem 7 *Every progressive, single-version TM implementation that ensures opacity and uses invisible reads has the time complexity of $\Omega(k)$, where $k = |Obj|$.*

The intuition behind the proof is the following. 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.

Proof. Assume that the only shared objects are registers $r_1, \ldots, r_k$. By contradiction, assume that there is a progressive, single-version transactional memory implementation that ensures opacity, uses invisible reads and has the worst-case time complexity of an operation of $o(k)$.

Let E be a low-level history corresponding to the following execution, in which only processes p_1 and p_2 take steps:

Phase 1. Transaction T_1, executed by process p_1, writes to all shared registers some values (known only to T_1), and commits.

Phase 2. Transaction T_2, executed by process p_2, performs read operations on $\Theta(k)$ different registers from some set Q_2 known only to T_2.

Phase 3. Transaction T_3, executed by process p_1, performs write operations on $\Theta(k)$ different registers from some set Q_3 known only to T_3, such that $Q_2 \cap Q_3 = \emptyset$, changing the state of all the written registers. Then T_3 commits.

Phase 4. Transaction T_2 reads a register $r \in Q_3$ and commits.

Claim 8 *$E|H$ is a valid history of I.*

Proof. To prove the claim, we show that none of the transactions in E can be forcefully aborted. Firstly, transactions T_1 and T_3 cannot be forcefully aborted, because I is progressive. (Note that there is no conflict between T_2 and T_3 until Phase 4, at which T_3 is already committed.) Secondly, T_2 has only one conflict, and the conflicting transaction of T_2 is T_3. However, in no prefix of $E|H$, in which T_3 is a conflicting transaction of T_2, T_3 is live. Therefore, as I is progressive, transaction T_2 cannot be forcefully aborted. □

Low-level history E is of the form $E_{\mathrm{p}} \cdot E_{\mathrm{m}} \cdot E_2$, where E_{p} corresponds to Phases 1 and 2, E_{m}—to Phase 3, and E_2—to Phase 4. In E_{p}, T_2, executed by p_2, is the only live transaction. By the invisible reads and no background threads properties of I, T_2 is invisible in E_{p}. Sequence E_{m} does not contain any step of process p_2, and sequence E_2 consists of only steps and events of process p_2, because of the no background threads property of I. Also, by our assumption, the number of steps process p_2 executes in E_2 is $o(k)$. Therefore, by the limited capacity property of I, there exists a set Q of shared objects, of size $|Obj| - o(k)$, such that for every function $c : Q \to Q$ and for every $E'_{\mathrm{m}} \sim_c E_{\mathrm{m}}$, if $E_{\mathrm{p}} \cdot E'_{\mathrm{m}}$ is a valid low-level history of I, then $E_{\mathrm{p}} \cdot E'_{\mathrm{m}} \cdot E_2$ is also a valid low-level history of I.

Claim 9 *For sufficiently large k, there is a function $c : Q \to Q$ and a sequence E'_{m}, such that (1) $E'_{\mathrm{m}} \sim_c E_{\mathrm{m}}$, (2) $E'_{\mathrm{m}}|H$ is the same as $E_{\mathrm{m}}|H$, except for one operation execution which in $E'_{\mathrm{m}}|H$ accesses a register r' from set Q_2, instead of a register $r'' \in Q_3$, $r'' \neq r$, and (3) $E_{\mathrm{p}} \cdot E'_{\mathrm{m}}$ is a valid low-level history of I.*

Proof. Firstly, the size of set Q is $k - o(k)$. This means that for sufficiently large k, there are some registers $r' \in Q_2$ and $r'' \in Q_3$, $r'' \neq r$, that are both in set Q, for the size of both Q_2 and Q_3 is $\Theta(k)$. Let $c : Q \to Q$ be the function, such that: (1) $c(r') = r''$, (2) $c(r'') = r'$, and (3) $c(r) = r$ if $r \neq r'$ and $r \neq r''$. Clearly, for such a function c there exists a low-level history E'_{m}, for which conditions (1) and (2) are satisfied.

Transactional memory cannot restrict the operations performed by a transaction in any other way than by (forcefully) aborting the transaction. The only transaction that is issuing events in E'_{m} is T_3. Thus, to prove that condition (3) is satisfied for some sequence E'_{m}, which satisfies (1) and (2), we only need to show that transaction T_3 cannot be forcefully aborted in low-level history $E' = E_{\mathrm{p}} \cdot E'_{\mathrm{m}}$. Indeed, E' is indistinguishable for process p_1 from a low-level history E'', in which transaction T_2 does not issue any operation invocation events. That is because all operations performed by T_2 are read-only and I uses invisible reads. But in E'' transaction T_3, executed by p_1, cannot be forcefully aborted, because I is progressive. Thus, T_3 cannot be forcefully aborted in E'. □

From Claim 9 and the limited capacity property of I we have that sequence $E' = E_{\mathrm{p}} \cdot E'_{\mathrm{m}} \cdot E_2$ is a valid low-level history of I. We will lead to a contradiction by proving, in the following claim, that E' violates opacity.

Claim 10 *Low-level history E' is not opaque.*

Proof. By the single version property of I, the value returned to T_2 in Phase 4 must be the value written to register r by transaction T_3. However, in E', T_2 must have already read from register r' a value written by T_1, not the value written by T_3. That is because T_2 returned from the read operation on r' before T_3 started.

Assume, by contradiction, that $E'|H$ is opaque. Consider $H' = nonlocal(E'|H) = E'|H$. By Theorem 2, there exists a total order $\ll$ on the set of transactions in H', such that $OPG(H', \ll, \emptyset)$ is well-formed and acyclic (note that there is no commit-pending transaction in H'). Regardless relation $\ll$, there are the following edges in the opacity graph:

- $T_1 \xrightarrow{\text{rt}} T_2$ and $T_1 \xrightarrow{\text{rt}} T_3$ (because T_1 precedes both T_1 and T_2),
- $T_1 \xrightarrow{\text{rf}} T_2$ and $T_3 \xrightarrow{\text{rf}} T_2$ (because T_2 reads all registers from set Q_2 from T_1, and register r from T_3).

We have two possibilities concerning total order $\ll$:

1. If $T_2 \ll T_3$, then there is an edge $T_2 \xrightarrow{\text{rw}} T_3$, because $T_2 \ll T_3$, T_2 reads r' and T_3 writes r'. Hence, there is a cycle $T_2 \xrightarrow{\text{rw}} T_3 \xrightarrow{\text{rf}} T_2$.
2. If $T_3 \ll T_2$, then there is an edge $T_3 \xrightarrow{\text{ww}} T_1$, because T_3 is committed, $T_3 \ll T_2$, T_3 writes to r', and T_2 reads r' from T_1. Hence, there is a cycle $T_3 \xrightarrow{\text{ww}} T_1 \xrightarrow{\text{rt}} T_3$.

Thus, for every possible total order $\ll$ among transactions T_1, T_2 and T_3, graph $OPG(H', \ll, \emptyset)$ has a cycle, and so $E'|H$ (and thus also E') is not opaque—a contradiction. □

□

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 worst-case time complexity of an operation $\Theta(k)$ (with most contention managers). It is worth noting that TL2 has a constant time complexity of an operation, 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.

8.3 Non-opaque TM Implementation with Constant Time Complexity

An example implementation of a progressive, single-version TM that uses invisible reads and ensures global atomicity (with real-time ordering) and strict recoverability, and that has constant time complexity, is presented in Algorithm 1. The algorithm is similar to the one of TL2 [6]: every shared object accessed by a transaction with a non-read-only operation is locked, and the new state of the object is not written to shared memory (array M) until the transaction invokes *tryC*. When a shared object is accessed with a read-only operation, its state is stored locally at the process executing the transaction, and the state is re-read and validated at commit time. Clearly, the algorithm works only in systems in which no process can crash.

Each shared object *ob* is mapped to two base shared objects: $M[ob]$, storing the current state and version (timestamp) of *ob*, and $L[ob]$, storing the lock that needs to be acquired by every process that wants to modify $M[ob]$. Additionally, the algorithm uses a global counter V for generating unique and monotonically increasing timestamps. The algorithm uses function *isReadOnly* to determine whether a given operation is read-only. We also assume that the following locking-related functions are implemented outside the algorithm: (1) *lock*, which acquires a given lock, if it is not acquired by any process, and returns *true*, or returns *false* otherwise (i.e., it does not block waiting until the lock is released), (2) *unlock*, which releases a given lock, and (3) *isLocked*, which returns *true* if a given lock is acquired by some process and *false* otherwise. We use the notation: $state.op(args)$ to denote the action of performing operation *op* with arguments *args* on a shared object in state *state*. We assume that such an action returns the new state of the object and the value returned by *op*.

The main differences between our algorithm and TL2 are the following. Firstly, as our algorithm uses a weaker correctness criterion than opacity, it can defer validation of read-only operations until a transaction invokes *tryC*. Clearly, this means that inconsistent state might be accessed by transactions (though these will be aborted later). Secondly, because our algorithm is progressive, unlike TL2, it cannot use the read timestamp of a transaction T_i (i.e., the value of V when T_i issues its first event) to validate operations issued by T_i. Instead, our algorithm checks, after T_i invokes $tryC_i$, whether any of the timestamps of shared objects accessed by T_i has changed. If yes, T_i is aborted. Thirdly, for simplicity reasons (our algorithm is devised for the sole purpose of complementing the lower bound proof) we assume that both the state of a shared object and a timestamp can be stored in a single base shared object.

Theorem 11 *Algorithm 1 implements a progressive, single-version transactional memory that uses invisible reads, satisfies global atomicity and strict recoverability, and preserves real-time ordering of transactions.*

Proof. *(sketch)* Let us denote by I the implementation shown in Algorithm 1. We divide the proof into five parts. Firstly, we prove that I guarantees global atomicity. Secondly, we show that I always preserves real-time order of transactions. Thirdly, we prove that I satisfies strict recoverability. Fourthly, we show that I is progressive. Finally, we show that I satisfies the assumptions we made for TM implementations, is single-version and uses invisible reads.

Global atomicity. Let us observe first that a process executing a transaction T_k can only write to array M when T_k is commit-pending and when T_k can no longer be aborted (i.e., after line 26 of the algorithm). Therefore, no transaction can observe any modifications to shared objects done by transactions that are live, and not commit-pending, or aborted. As global atomicity does not put any requirement on non-committed transactions, we can consider only histories in which every transaction is either commit-pending or committed.

Let E be any low-level history of I, and E_c be the longest subsequence of E that contains only events and steps of transactions that (1) are committed, or (2) are commit-pending and can no longer be aborted (i.e., their corresponding process executed already lines 20–25 of the algorithm). Let h be the longest subsequence of E_c that contains only steps executed in line 13 (reading the value of $M[ob]$) or in line 30 (writing to $M[ob]$). It is straightforward that if h is serializable (in the classical sense of conflict serializability; see, e.g., [10]), then $E|H$ satisfies global atomicity (after events of live and aborted transactions are removed).

To simplify the discussion, we will say that a transaction T_i executes a read (write) on $M[ob]$, instead of saying that the process executing T_i executes a read (write) instruction on base shared object $M[ob]$ in h. We will also say that a transaction T_i is in h, meaning that some steps corresponding to T_i are in h. It is worth noting that each transaction first executes all reads, and then all writes (if any).

Algorithm 1: An implementation of a progressive, single-version transactional memory that uses invisible reads and ensures global atomicity (with real-time ordering) and strict recoverability

```
1  upon $inv_i(ob, op, args)$ do
2      if $ob \in wset$ then $(state, ts) \leftarrow localcopy[ob]$;
3      else
4          if not $isReadOnly(op)$ then
5              $locked \leftarrow L[ob].lock()$;
6              if not $locked$ then
7                  $reset()$;
8                  return $A_k$;
9          else if $L[ob].isLocked()$ then
10             $reset()$;
11             return $A_k$;
12         if $ob \in rset$ then $(state, ts) \leftarrow localcopy[ob]$;
13         else $(state, ts) \leftarrow M[ob]$;
14     $(newstate, val) \leftarrow state.op(args)$;
15     $localcopy[ob] \leftarrow (newstate, ts)$;
16     if $isReadOnly(op)$ then $rset \leftarrow rset \cup \{ob\}$;
17     else $wset \leftarrow wset \cup \{ob\}$;
18     return $ret_i(ob, op, val)$;
19 upon $tryC_i$ do
20     foreach $ob \in rset$ do
21         $(state, ts) \leftarrow localcopy[ob]$;
22         $(curstate, curts) \leftarrow M[ob]$;
23         if $L[ob].isLocked()$ or $curts > ts$ then
24             $reset()$;
25             return $A_k$;
26     if $wset \neq \emptyset$ then
27         $wts \leftarrow V.inc()$;
28         foreach $ob \in wset$ do
29             $(state, ts) \leftarrow localcopy[ob]$;
30             $M[ob] \leftarrow (state, wts)$;
31     $reset()$;
32     return $C_k$;
33 upon $tryA_i$ do
34     $reset()$;
35     return $A_k$;
36 procedure $reset()$
37     foreach $ob \in wset$ do $L[ob].unlock()$;
38     $rset \leftarrow wset \leftarrow \emptyset$;
```

We say that transaction T_i writes after transaction T_k writes, if T_i executes a write on $M[ob]$ after T_k executes a write on $M[ob]$, in h, for some shared object ob. In the same way we can define when T_i writes after T_k reads, and when T_i reads after T_k writes.

A serialization graph of h is a graph in which each transaction in h is a vertex of the graph, and there is an edge from vertex T_i to vertex T_k if (1) T_k writes after T_i writes, or (2) T_k reads after T_i writes, or (3) T_k writes after T_i reads. The theory of serializability says that history h is conflict serializable if the corresponding serialization graph is acyclic.

Assume, by contradiction that the serialization graph of h is not acyclic. That is, there is a cycle C in the graph. Let T_i be the transaction from cycle C that executes the last write. Let T_k be the transaction that follows T_i in cycle C. We will lead to a contradiction by showing that it is not possible that all transactions in C commit.

Assume first that T_k is not read-only, i.e., T_k invokes at least one non-read-only operation on some shared object. Clearly, T_k cannot write after T_i writes, because then all writes of T_k would have to follow every write of T_i, contradicting the assumption that T_i is the transaction from C that executes the last write. That is because the process executing transaction T_i locks all the shared objects accessed by transaction T_i with a non-read-only operation (line 5) before writing to any of the corresponding base shared object $M[ob]$ (line 30), and unlocks them only just before committing T_i (line 37, *reset* called in line 31). Thus, either (1) T_k reads after T_i writes, or (2) T_k writes after T_i reads.

If T_k reads some $M[ob]$ after T_i writes $M[ob]$, then T_k observes that $L[ob]$ is locked in line 5 or 9. That is for the following reasons. Firstly, the read executed by T_k on $M[ob]$ must be between writes executed by T_i, for T_k is not read-only and its last write precedes the last write of T_i, and follows the last read of T_k. Secondly, $L[ob]$ must be locked by T_i from before its first write, until after its last write. Thus, T_k must be aborted in line 8 or 11—a contradiction.

If T_k writes some $M[ob]$ after T_i reads $M[ob]$, then T_i must read, in line 22, the value written to $M[ob]$ by T_k. That is because T_i invokes $tryC_i$ after executing its last write, which follows every write of T_k. Thus, T_i observes, in line 23, that $curts > ts$ (for a timestamp of an object can never decrease) and aborts in line 25—a contradiction.

Therefore, transaction T_k is read-only, and T_k reads some $M[ob]$ after T_i writes $M[ob]$. In fact, T_k reads $M[ob]$ after T_i performs all writes and unlocks $M[ob]$. Otherwise, T_k would observe that $L[ob]$ is locked in line 9 and abort.

Let T_m be the transaction that follows T_k in cycle C. Clearly, T_m writes after T_k reads. Moreover, the last write of T_m must be before the last read of T_k, which follows the last write of transactions in C. Therefore, T_k must observe in line 22 the value written by T_m, and abort—a contradiction.

Real-time order. Assume, by contradiction that there is a history H in which real-time order is not preserved. That is, there are two transactions T_i and T_k in H, such that $T_i \prec_H T_k$, but there is a path C from T_k to T_i in the serialization graph of corresponding history h (constructed from H in the same way as in the global atomicity part of the proof). But then we can follow the same argument as when proving global atomicity and show that some transaction in C must be aborted—a contradiction.

Strict recoverability. In our model, strict recoverability can be defined (informally) as follows: If a transaction T_i changes the state of a shared object *ob* to some value s, different then the last committed state of *ob*, then no other transaction observes that *ob* is in state s until T_i commits. By T_i *commits* we mean the point in time at which updates done by T_i to shared objects become visible to other transactions, not the point in time at which T_i receives a commit event.

Clearly, I satisfies strict recoverability: Every transaction that is to modify some shared object first locks the object, and does not unlock any object until all updates are performed. Moreover, if another transaction tries to execute an operation on a locked object, the transaction will get aborted, and thus will not be able to observe the state of the locked object.

Progressive. A transaction can be forcefully aborted in one of the following two cases. Firstly, a process p_i can forcefully abort a transaction T_k (executed by p_i) that invoked $inv_k(ob, op, args)$, if p_i detects that *ob* is already locked by some other process p_j, executing a transaction T_m. But then T_m is a conflicting transaction of T_k, and T_m must be live (and conflicting with T_k) at the point when p_i checks the corresponding lock. Secondly, p_i can forcefully abort T_k when T_k invokes $tryC_k$, and some shared object in set *rset* either is locked (which is the same as the previous case), or has been updated since T_k accessed this object with a read-only operation. But in the latter case there must have been a transaction T_m that was live, and that updated some shared object *ob*, at some point after T_k accessed the object *ob*. Thus, I is progressive.

Other properties. In every pair of base shared objects $M[ob]$ and $L[ob]$ only information about shared object *ob* is stored. Thus, limited capacity is satisfied. The single-version and no background threads properties are also satisfied by I: this is clear from the algorithm.

When $inv_i(ob, op, args)$ is invoked and *op* is a read-only operation, no lock is acquired in line 5. Also, shared object *ob* is not added to set *wset* in line 17, and so base shared object $M[ob]$ is not modified later, in line 30, when the transaction tries to commit (unless some non-read-only operation is invoked on *ob* by the transaction). Thus, I satisfies the invisible reads property. □

9 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.[15] 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 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.

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*

[15]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.

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

[11] 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.

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

[13] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, Jan 1991.

[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.