

# The Cost of Privatization\*

Hagit Attiya  
Technion & EPFL

hagit@cs.technion.ac.il

Eshcar Hillel  
Technion

eshcar@cs.technion.ac.il

July 28, 2010

## Abstract

*Software transactional memory* (STM) guarantees that a *transaction*, consisting of a sequence of operations on the memory, appears to be executed atomically. In practice, it is important to be able to run transactions together with *nontransactional legacy* code accessing the same memory locations, by supporting *privatization*. Privatization should be provided without sacrificing the parallelism offered by today's multicore systems and multiprocessors.

This paper proves an inherent cost for supporting privatization, which is linear in the number of privatized items. Specifically, we show that a transaction privatizing  $k$  items must have a data set of size at least  $k$ , in an STM with invisible reads, which is oblivious to different non-conflicting executions and guarantees progress in such executions. When reads are visible, it is shown that  $\Omega(k)$  memory locations must be accessed by a privatizing transaction, where  $k$  is the minimum between the number of privatized items and the number of concurrent transactions guaranteed to make progress, thus capturing the tradeoff between the cost of privatization and the parallelism offered by the STM.

## 1 Introduction

*Software transactional memory* (STM) is an attractive paradigm for programming parallel applications for multicore systems. STM aims to simplify the design of parallel systems, as well as improve their performance with respect to sequential code by exploiting the scalability opportunities offered by multicore systems. An STM supports *transactions*, each encapsulating a sequence of operations applied on a set of *data items*; an STM guarantees that if any operation takes place, they all do, and that if they do, they appear to do so atomically, as one indivisible operation.

In practice, some operations cannot, or simply are preferred not to be executed within the context of a transaction. For example, an application may be required to invoke irrevocable operations, e.g., I/O operations, or use library functions that cannot be instrumented to execute within a transaction. *Strong atomicity* [20, 23, 29] guarantees isolation and consistent ordering of transactions in the presence of non-transactional memory accesses. Supporting strong atomicity is crucial both for interoperability with legacy code and in order to improve performance.

A simple solution is to make each nontransactional operation a (degenerate) transaction, but this means that nontransactional operations incur the overhead associated with a transaction. Although compiler optimizations can reduce this cost in some situations [3, 28], they do not alleviate it completely. Thus, STMs

---

\*This research is supported in part by the *Israel Science Foundation* (grant number 953/06).

seek to improve performance by supporting *uninstrumented* nontransactional operations [14, 31], which are executed as is, typically as a single access to the shared memory.

Many recent STMs [8, 9, 11, 13, 21, 22, 24, 25, 32] provide strong atomicity by supporting *privatization* [29, 31], thereby allowing to “isolate” some items making them private to a process; the process can thereafter access them nontransactionally, without interference by other processes. It is commonly assumed that privatizing a set of items simply involves disabling all shared references to those items [9, 21, 32], e.g., by nullifying these references. However, it has been claimed that privatization is a major source of overhead for transactional memories [34], and that supporting uninstrumented nontransactional operations can seriously limit their parallelism [7].

Consider, for example, the linked-list depicted in Figure 1, in which every node points to a root item. Every root item points to some disjoint subgraph, such as a tree, and is the only path to items in the subgraph. Throughout the paper we consider a *workload* in which one transaction privatizes all root items and their subgraphs, and other transactions read all the nodes of the linked list but write only to one root item that is pointed from the list. In this workload, the hope is that a constant amount of work, e.g., nullifying the head of the linked-list, will suffice for privatizing the whole linked list; afterwards, all items in the rooted trees can be accessed by the privatizing process with simple (uninstrumented) reads and writes to the shared memory.

This paper proves that in many important workloads, including the linked list presented above, the hope to combine efficient privatizing transactions with uninstrumented nontransactional reads, cannot be realized, unless parallelism is compromised. Specifically, the privatizing transaction must incur an inherent cost, linear in the number of data items that are privatized and later accessed with uninstrumented reads.

Our lower bounds do not apply to overly sequential STMs, which achieve efficient privatization by using a single global lock and allowing only one transaction to make progress at each time [8, 25], thereby significantly reducing the throughput. We make a fairly weak progress assumption (Property 1), requiring the STM to allow concurrent progress of nonconflicting transactions: a transaction can abort or block only due to a conflicting pending transaction.

We show that *eager* STMs, in which a transaction may update the memory before it is guaranteed to commit cannot support privatization, under this weak progress assumption (Theorem 1 in Section 2).

A key factor in many efficient STMs is not having to track the data sets of other transactions, especially if they are not conflicting. We capture this feature by assuming that the STM is *oblivious*, namely, a transaction does not distinguish between nonconflicting transactions (Property 2). A simple example is provided by STMs using a global clock or counter [10, 26, 27], or a decentralized clock [5], in which a transaction cannot tell whether a process  $p$  executes a transaction that writes to item  $x$  or a transaction that writes to item  $y$ , unless it accesses either  $x$  or  $y$ ; it can observe that the clock or a counter has increased, but this happens in both cases. A less-immediate example is the behavior of TLRW [11] for so-called *slotted* threads. Several other STMs [8, 9, 13, 24, 32] are also oblivious. (A detailed discussion appears in Section 7.)

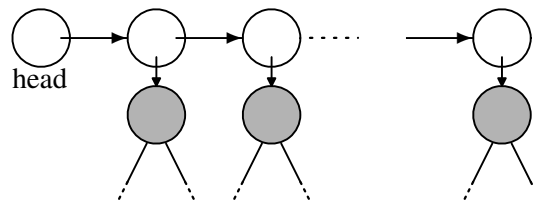


Figure 1: In this example, the privatizing transaction sets the head to NULL and privatizes the dark items and their subgraphs. Other transactions write to the dark items; each transaction writes to a different one.

Our first main result further assumes that reads do not write to the memory (*invisible* reads) and shows that a transaction privatizing  $k$  items must have a data set of size  $\Omega(k)$  (Theorem 2 in Section 3). In an oblivious STM with invisible reads, transactions are unaware of, and hence, unaffected by, read-read conflicts. In the linked-list example this means that, for every process, the execution of other transactions appears only to write to a single item (either the head of the list or an item pointed by the links).

Our second main result removes the assumption of invisible reads, and shows an  $\Omega(k)$  lower bound on the number of shared memory accesses performed by a privatizing transaction, where  $k$  is the minimum between the number of privatized items and the level of parallelism, i.e., the number of transactions guaranteed to make progress concurrently (Theorem 4 in Section 4). The proof is more involved and relies on the assumption that the STM provides a significant level of parallelism. This lower bound explains why the *quiescence* mechanism [9, 24, 31], for example, must compromise parallelism in order to support efficient privatization.

Obliviousness generalizes *disjoint-access parallelism* [19], and our lower bounds hold also for disjoint-access parallel STMs (see Section 5).

Our proofs only assume a weak safety property that requires a nontransactional read of a data item, where no nontransactional write precedes the read, to return the value written by an earlier committed transaction, or the initial value, if no such transaction commits. This property follows from *parameterized opacity* [14], regardless of the memory model imposed on nontransactional reads and writes. We show that STMs allowing more than one transaction to proceed concurrently, without using any privatization mechanism, cannot be opaque (Theorem 7 in Section 6).

We start in Section 2, with basic definitions and description of various STM properties needed for our results. Section 3 includes the lower bound on the size of the data set of privatizing transactions in STMs with invisible reads. In Section 4, we bound the cost of privatization with visible reads, and sketch an STM that matches the lower bound. Section 5 presents the results for disjoint-access parallel STMs. We briefly discuss STMs that allow uninstrumented operations on items without prior privatization (Section 6), and review other related work (Section 7). We conclude with a discussion of the results and directions for further work, in Section 8.

## 2 Preliminaries

A *transaction* is a sequence of operations executed by a single process on a set of *data items* shared with other transactions; each item is initialized to some initial value. The collection of data items accessed by a transaction is the transaction's *data set*; in particular, the items written by the transaction are its *write set*, and the items read by the transaction are its *read set*.

A *software implementation of transactional memory (STM)* provides data representation for transactions and data items using *base objects*, and algorithms, specified as *primitive operations* (abbreviated *primitives*) on the base objects, which *asynchronous* processes have to follow in order to execute the operations of transactions. In addition to ordinary read and write primitives, we allow *arbitrary* read-modify-write primitives, including CAS. A primitive is *nontrivial* if it may change the value of the object, e.g., a write or CAS; otherwise, it is *trivial*, e.g., a read. An *access* to base object  $o$  is the application of a primitive to  $o$ .

An *event* is a computation *step* by a process consisting of local computation and the application of a primitive to base objects, followed by a change to the process's state, according to the results of the primitive. A *configuration* is a complete description of the system at some point in time, i.e., the state of each process and the state of each shared base object. In the unique *initial* configuration, every process is in its initial state and every base object contains its initial value.

Two executions  $\alpha_1$  and  $\alpha_2$  are *indistinguishable* to a process  $p$ , if  $p$  goes through the same sequence of state changes in  $\alpha_1$  and in  $\alpha_2$ ; in particular, this implies that  $p$  goes through the same sequence of events.

## 2.1 STM Properties

The consistency condition assumed by all our results is that if a transaction writing to an item  $t$  a value other than the initial value, commits, then a later nontransactional read of  $t$  returns a value that is different from the initial value; vice versa, if no transaction writing to  $t$  commits and no nontransactional write changes  $t$  then a nontransactional read of  $t$  returns the initial value. This condition is satisfied by *parameterized opacity* [14] (see Section 6), and hence our lower bounds hold for parameterized opacity as well.

A transaction *blocks* if it takes an infinite number of steps without committing or aborting. Our progress condition requires a transaction to commit if it has no nontrivial conflict<sup>1</sup> with any pending transaction; that is, a transaction can abort or block only due to a nontrivial conflict with such a transaction. A transaction  $T$  is *logically committed* in a configuration  $C$  if  $T$  does not abort in any infinite extension from  $C$ .

**Property 1 (*l*-progressive STM)** *An STM is  $l$ -progressive,  $l \geq 0$ , if a transaction  $T$  aborts or blocks in a solo execution (of all the transaction or a suffix of it) after an execution  $\alpha$  that contains  $l$  or less incomplete transactions, only due to a nontrivial conflict with an incomplete logically committed transaction.*

Note that a transaction that must commit according to this definition becomes logically committed at some point, e.g., right before it commits. Property 1 means that, in the absence of conflicts, the STM must ensure parallelism. This property (for any  $l \geq 1$ ) is satisfied by *weakly progressive* STMs [16], in which a transaction must commit if it does not encounter conflicts, and by *obstruction-free* STMs [17], in which a transaction commits when it runs by itself for long enough, implying that it must not abort or block if it runs solo after an execution without nontrivial conflicts.

An  *$l$ -independent* execution contains  $l \geq 0$  transactions, each executed by a different process,  $p_{i_1}, \dots, p_{i_\ell}$ , running solo until it is logically committed, on data sets without nontrivial conflicts. An STM is *oblivious* if a transaction running solo after an  $l$ -independent execution, without nontrivial conflicts with the pending transactions, behaves in a manner that is independent of the data sets of the pending transactions.

**Property 2 (Oblivious STM)** *An STM is oblivious if for any pair of  $l$ -independent executions  $\alpha_1$  and  $\alpha_2$ , each containing  $l$  transactions executed by the same processes  $p_{i_1}, \dots, p_{i_\ell}$ , in the same order, if some transaction  $T$  executed by a process  $p$  does not have nontrivial conflicts with the transactions in  $\alpha_1$  and  $\alpha_2$ , then  $\alpha_1 T$  and  $\alpha_2 T$  are indistinguishable to  $p$ .*

An STM has *invisible reads*, if an execution of any transaction is indistinguishable from the execution of a transaction writing the same values to the same items, while omitting all read operations. More formally, consider an execution  $\alpha$  that includes a transaction  $T$  of process  $p$  with write set  $W$  and read set  $R$ , and consider a transaction  $T'$  of process  $p$  writing the same values to  $W$  in the same order as in  $T$ , but with an empty read set. In STMs with invisible reads there is an execution  $\alpha'$  that includes  $T'$  instead of  $T$ , such that  $\alpha'$  is indistinguishable to all other processes from  $\alpha$ .

<sup>1</sup>A *conflict* occurs when two operations access the same data item; the conflict is *nontrivial* if one of the operations is a write.

## 2.2 Privatization

An STM may contain transactions that privatize a set of data items. Rather than getting into the details of what privatization means, we only state a property that is naturally expected out of any notion of privatization, as it guarantees isolation when accessing the shared memory with nontransactional operations.

We assume *uninstrumented* nontransactional read operations, which simply read a fixed base object. The base object might depend on the process and the item, e.g., a private (local) copy of the item, but the process applies no manipulation on the value and simply returns the value written in the base object as the value of the item.

Process  $p_j$  *privatizes* item  $t_i$  when  $p_j$  commits a transaction privatizing  $t_i$ . The private base object that process  $p_j$  associates with a data item  $t_i$ , after privatizing the item, is denoted  $m_i^j$ . Uninstrumented nontransactional write operations can be defined analogously (although they are not used in our proofs).

**Property 3 (Privatization-safe STM)** *An STM with uninstrumented nontransactional operations is privatization safe if after process  $p_j$  privatizes item  $t_i$ , no process  $p_h \neq p_j$  applies a nontrivial primitive to the base object  $m_i^j$ .*

Previous work [21] informally assumed that a transaction privatizing a region must conflict with other transaction accessing this region.

Indeed, it can be easily shown that a weakly progressive STM cannot support privatization *if the read set of every writing transaction is empty*, unless the privatizing transaction accesses all the items it privatizes. If there is an item the privatizing transaction does not access, then a transaction writing to this item executed after the privatizing transaction completes, is unaware of the privatization, and may access private locations.

We first show that in a privatization-safe STM, a transaction applies a nontrivial primitive (e.g., writes) to a base object associated with a privatized item, only after it is already logically committed.

An STM is *eager* if there exists a configuration  $C$  such that a transaction  $T$  is the only pending transaction in  $C$ ,  $T$  is not logically committed, and a process  $p$  executing  $T$  applies a nontrivial primitive to a base object associated with an item that another process privatizes. It is simple to show that a 1-progressive eager STM is not privatization-safe. Note that assuming uninstrumented nontransactional operations implies that the mapping of each privatized item to the corresponding base object is static.

**Theorem 1** *An uninstrumented 1-progressive eager STM is not privatization-safe.*

**Proof:** Since the STM is uninstrumented, let  $m_i^0$  be the private base object which process  $p_0$  associates with item  $t_i$ . Assume another process  $p_1$  executing a transaction  $T_1$  applies a nontrivial primitive to  $m_i^0$  before  $T_1$  is logically committed in some configuration  $C$ , where only  $T_1$  is pending; call this event  $\tau$ . Consider a transaction  $T_0$  of  $p_0$  privatizing the item  $t_i$ , executed from  $C$ .

Since only  $T_1$  is pending in  $C$ ,  $\alpha$ , the execution leading to  $C$ , has no incomplete logically committed transaction conflicting with  $T_0$ . Since the STM is 1-progressive,  $T_0$  completes successfully when executed after  $\alpha$ , and since the STM is uninstrumented,  $m_i^0$  is private to  $p_0$  after  $\alpha T_0$ . However,  $\tau$  can be applied to  $m_i^0$ , even after  $T_0$ , in contradiction to privatization safety. ■

In the sequel, we assume that a privatization-safe STM is not eager.

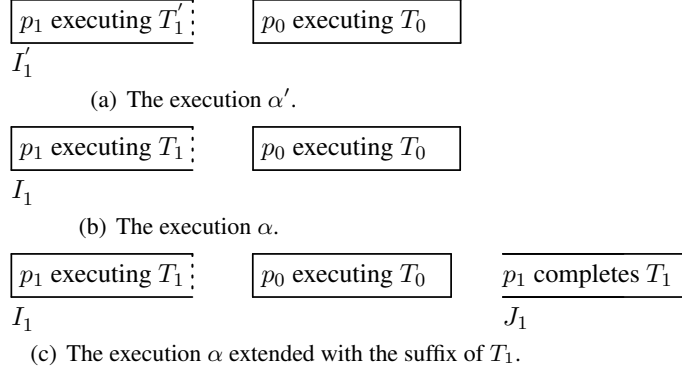


Figure 2: The executions used in the proof of Theorem 2. A dotted line indicates that the transaction is logically committed.

### 3 Privatization with Invisible Reads

The next theorem shows that in an oblivious STMs supporting privatization, the data set of a privatizing transaction must contain all privatized items. The proof proceeds by creating a scenario in which a privatizing transaction misses the up-to-date value of a privatized item; some care is needed in order to argue about each item separately.

**Theorem 2** *For any privatization-safe STM that is 1-progressive, oblivious and with invisible reads, there is a privatization workload in which transactions have nonempty read sets, for which there is an execution where the size of the data set of a transaction privatizing  $k$  items is  $\Omega(k)$ .*

**Proof:** Consider two processes  $p_0$  and  $p_1$ :  $p_0$  executes a transaction  $T_0$  that privatizes the items  $t_1, \dots, t_k$ . For  $p_1$ , consider a transaction  $T_1'$  with an arbitrary read set, writing to an item  $u$  that is never accessed by  $T_0$ .

Consider the execution  $\alpha' = I_1' T_0$ , such that in  $I_1'$ ,  $p_1$  executes a prefix of the transaction  $T_1'$  until it is logically committed (see Figure 2(a)).  $I_1'$  is indistinguishable to  $p_0$  from an execution in which  $p_1$  executes a transaction that only writes to  $u$  until it is logically committed. After  $I_1'$ , there is no incomplete transaction that has a conflict with  $T_0$ , and since the STM is 1-progressive,  $T_0$  commits when executed after  $I_1'$ .

Assume, by way of contradiction, that the data set of  $T_0$  does not include some item  $t_i$  that it privatizes when executed after  $I_1'$ . Consider the execution  $\alpha = I_1 T_0$ , such that in  $I_1$ ,  $p_1$  executes a prefix of a transaction  $T_1$  with the same read set as  $T_1'$  and writing to the item  $t_i$  a value different than its initial value, and  $T_1$  is logically committed after  $I_1$  (see Figure 2(b)). It can be shown (Lemma 8 in the appendix) that  $t_i$  is not in the data set of  $T_0$  also when executed after  $I_1$ .

Since the reads are invisible  $I_1$  is indistinguishable to  $p_0$  from an execution with no nontrivial conflicts, and since the STM is oblivious,  $T_0$  commits also when executed after  $I_1$  in  $\alpha$ . Let  $m_1, \dots, m_k$  be the base objects that are private to  $p_0$  after  $\alpha$  (we omit the superscript 0). Since the STM is 1-progressive  $T_1$  commits when completed after  $\alpha$ . Since  $T_1$  is logically committed after  $I_1$ , it writes to  $t_i$ . Consider the execution  $I_1 T_0 J_1$ , such that  $J_1$  is the suffix of the execution of  $T_1$  until it commits (see Figure 2(c)).

**Lemma 3**  $p_1$  modifies the state of  $m_i$  in  $J_1$ .

**Proof:** We first show that  $p_1$  does not modify  $m_i$  in the first part of its transaction in  $\alpha$ . Assume that  $p_1$  applies a nontrivial primitive to  $m_i$  in some step when executing  $I_1$  in  $\alpha$ , and let  $\tau$  be the first such step.

Let  $\widehat{I}_1$  be the prefix of  $I_1$  preceding  $\tau$ .  $I_1$  is the shortest prefix of  $T_1$  after which  $T_1$  is logically committed. Hence,  $T_1$  is not logically committed after  $\widehat{I}_1$ , and it is the only transaction that is pending after  $\widehat{I}_1$ . In a solo execution of  $T_0$  after  $\widehat{I}_1$ ,  $T_0$  is committed, making  $m_i$  private to  $p_0$ . Since the STM is not eager,  $p_1$  does not apply  $\tau$  after  $\widehat{I}_1$ .

A similar argument shows that  $p_1$  does not apply nontrivial primitive to  $m_i$  in  $\alpha'$ .

Next, we argue that  $p_0$  does not modify the state of  $m_i$  in  $\alpha$ . Otherwise, a nontransactional, uninstrumented read operation, to  $t_i$  of  $p_0$  after  $\alpha'$  returns a value that is not the initial value of  $m_i$ , whereas no committed transaction writes to  $t_i$  in  $\alpha'$ , contradicting our correctness condition. The executions of  $T_0$  after  $I_1$  and  $I'_1$  are indistinguishable, since the STM is oblivious and since  $T_0$  does not access neither  $u$  nor  $t_i$ . We have shown that  $p_1$  accesses  $m_i$  neither in  $\alpha$  nor in  $\alpha'$ , and hence,  $p_0$  does not successfully apply a nontrivial primitive to  $m_i$  also in  $\alpha$ .

If  $p_1$  does not modify the state of  $m_i$  also in  $J_1$ , then a nontransactional read by  $p_0$  to  $t_i$  after  $I_1T_0J_1$  returns the initial value of  $m_i$ , since reads are uninstrumented. This contradicts our correctness condition since  $T_1$ , which writes to  $t_i$  a value that is different from the initial value of  $m_i$ , commits before the nontransactional read of  $p_0$ . ■

Therefore,  $p_1$  applies a nontrivial primitive to  $m_i$  in  $J_1$  after the execution of  $T_0$ , in contradiction to privatization safety. ■

The proof allows  $T_1$  and  $T'_1$  to have non-empty read sets. Since the reads are invisible, this looks to  $p_0$  as if they have empty read sets. Note however, that  $p_1$  does read from the memory and distinguishes its execution of  $T_1$  and  $T'_1$  from executing a transaction with an empty read set. Thus, the result does not follow from the trivial lower bound for transactions with empty read sets.

## 4 Privatization with Visible Reads

A similar lower bound holds also for STMs with visible reads, assuming they ensure some degree of parallelism. The cost is stated in terms of low-level accesses by the privatizing transaction, rather than in terms of the high-level aspects of the transaction. Some key ideas in the proof are similar to the proof of Theorem 2; however, the technical details are more involved, in order to handle visible reads.

**Theorem 4** *For any privatization-safe STM that is  $l$ -progressive and oblivious there is a privatization workload in which update transactions have nonempty read sets, for which there is an execution where a transaction privatizing  $m$  items accesses  $\Omega(k)$  base objects, where  $k = \min\{l, m\}$ .*

The proof of this theorem is quite involved technically, so we present it for a workload similar to the linked list of Figure 1; after the proof, we discuss how it can be extended to other scenarios. Before getting into the proof, we start with a high-level outline.

We have  $k$  updating transactions traverse the nodes of a linked list, while each transaction writes to a different item pointed by the list that is not read by other transactions; so these transactions have only trivial conflicts. Later, a transaction by another process, privatizing all items pointed by the linked list, is shown to miss the up-to-date value of the privatized items, unless it accesses many base objects.

Since reads are visible, however, it is difficult to hide the updating transaction from the privatizing transaction. The challenge is to create an execution in which an updating transaction runs long enough to guarantee that it will eventually commit—even after the privatizing transaction commits, and even if the

privatizing transaction writes to an item it reads—but not long enough to become visible to the privatizing transaction.

The privatizing transaction may write to an item in the read set of an updating transaction (e.g., the head of the list), thus invalidating its read set. Hence, to guarantee that an updating transaction eventually commits in the execution constructed, the updating transaction runs until it is logically committed, before the privatizing transaction starts.

It may seem that, at this point, the privatizing transaction does not need to access many objects to observe a conflict with the updating transactions, and it can abort or at least block until the conflicts are resolved. However, the obliviousness and non-eagerness of the STM can be used to “hide” the updating transactions from the privatizing transaction, by swapping the updating transactions with other *confusing* transactions, accessing a completely disjoint linked list; the confusing transactions also have only trivial conflicts among them. Due to obliviousness, these confusing transactions are indistinguishable from the original updating transactions.

We start with an execution in which the confusing transactions run one after the other; this execution is  $k$ -independent. Then, we swap confusing transactions with updating transactions. Swapping is done inductively: Each inductive step swaps one confusing transaction with an updating transaction by the same process; that is, at each step one additional process executes the updating transaction instead of the confusing transaction, and *incurs an access to at least one additional base object* by the privatizing transaction. This yields an execution in which the privatizing transaction accesses many objects, implying the lower bound.

Progressiveness is used to ensure that if at some point the privatizing transaction observes a conflict, the updating transaction causing the conflict may run to completion. This also ensures that the privatizing transaction runs to completion.

A technical challenge in the proof is in deciding which transaction to swap next, so as not to lose the accesses by the privatizing transaction that appear in the execution we have created so far. Specifically, we need to pick a transaction  $T$  such that swapping it is invisible to the privatizing transaction in its execution prefix, at least during the memory accesses incurred due to previous swaps. This is done by letting  $T$  be the last transaction to modify the next location seen by the privatizing transaction, so that future swaps will not overwrite locations  $T$  writes to and that are accessed by the privatizing transaction in its execution prefix. (This is the purpose of Item 5 maintained in the inductive construction.)

**Proof:** Consider the scenario described in Figure 3: Let  $r_0, r_1, \dots, r_k$  be the nodes of the linked list ( $r_0$  is the head node); each node  $r_i$ ,  $1 \leq i \leq k$  points to an item  $t_i$ . We consider  $k + 1$ ,  $k \geq 1$ , processes  $p_0, \dots, p_k$ . Process  $p_0$  executes a transaction  $T_0$  that privatizes the linked list, including the items  $t_1, \dots, t_k$ . Every process  $p_i$ ,  $1 \leq i \leq k$ , executes a transaction  $T_i$  that traverses the nodes of the list and then writes to  $t_i$  a value different than its initial value; namely, its read set is  $\{r_0, r_1, \dots, r_i\}$ , and its write set is  $\{t_i\}$ .

For the sake of the proof, we create a copy of the linked list, having exactly the same structure, which is not connected to the first list in any way. This list contains  $k + 1$  nodes  $r'_0, r'_1, \dots, r'_k$  and  $k$  items  $t'_1, \dots, t'_k$ .  $T_0$  does not access this list at all; however, for every process  $p_i$ ,  $1 \leq i \leq k$ , we take another transaction,  $T'_i$ , that traverses the nodes of the second list and then writes to  $t'_i$ ; namely, its read set is  $\{r'_0, r'_1, \dots, r'_i\}$ , and its write set is  $\{t'_i\}$ .

A process  $p$  reads from a process  $q$  via a base object  $o$  in an execution  $\alpha$  if  $p$  accesses  $o$ , and  $o$  was last modified by  $q$ . Process  $p$  reads from a set of processes  $P$  in an execution  $\alpha$  if for every process  $q \in P$ , there is a base object  $o$  such that  $p$  reads from  $q$  via  $o$  in  $\alpha$ .

Consider the following execution  $\alpha_0\beta_0\gamma_0$ :  $\alpha_0$  is  $I'_1 \dots I'_k$ , such that  $p_i$  executes in  $I'_i$ , a prefix of the transaction  $T'_i$  and  $T'_i$  is logically committed after  $I'_i$ ;  $\beta_0$  is the empty execution interval; and  $\gamma_0$  is a solo execution of  $T_0$  by  $p_0$  to completion (see Figure 4(a)).

For every  $\ell$ ,  $0 < \ell \leq k$ , we show how to perturb  $\alpha_{\ell-1}\beta_{\ell-1}\gamma_{\ell-1}$  to obtain an execution  $\alpha_\ell\beta_\ell\gamma_\ell$ , such that

1.  $p_0$  executes  $T_0$  to successful completion in  $\beta_\ell\gamma_\ell$ .
2.  $p_0$  reads from all processes in  $P_\ell$ , a subset of  $\{p_1, \dots, p_k\}$  of size (at least)  $\ell$ , in  $\alpha_\ell\beta_\ell$ .
3. There is a subset  $Q_\ell$  of  $P_\ell$ , where every process  $p_j \in Q_\ell$  executes  $I_j$ , a prefix of the transaction  $T_j$ , in  $\alpha_\ell$ , such that  $T_j$  is logically committed after  $I_j$ , and  $p_j$  completes  $T_j$  in  $\beta_\ell$ .
4. Every process  $p_j$ ,  $\{p_1, \dots, p_k\} \setminus Q_\ell$ , executes  $I'_j$  in  $\alpha_\ell$ .
5. For every process  $p_j$  from which  $p_0$  does not read in  $\alpha_\ell\beta_\ell\gamma_\ell$ ,  $\alpha_\ell^j$  is a  $k$ -independent execution, in which  $p_j$  executes  $I_j$  instead of  $I'_j$ , and all other processes take the same steps as in  $\alpha_\ell$ ; in  $\alpha_\ell^j$ ,  $p_j$  does not modify any base object  $o$ , such that, in  $\alpha_\ell\beta_\ell$   $p_0$  reads  $o$  from a process  $p_h$ ,  $h < j$ .

For  $\ell = k$ , we get an execution  $\alpha_\ell\beta_\ell\gamma_\ell$ , such that  $p_0$  reads from  $k$  different processes in  $P_k$  (Condition 2). The theorem follows since  $p_0$  accesses  $k$  different base objects,

The proof is by induction on  $\ell$ . We first show that all conditions hold for the base case,  $\ell = 0$ :

1. Since all the transactions in  $\alpha_0$  write to items that are not accessed by  $T_0$  in any execution and all the reads of the transaction in  $\alpha_0$  are from items not written by  $T_0$ , and since the STM is  $k$ -progressive,  $T_0$  completes successfully in  $\gamma_0$ .
2.  $\beta_0$  is an empty execution interval and  $p_0$  does not take any step in  $\alpha_0\beta_0$ . Hence,  $p_0$  does not read from any process in  $\alpha_0\beta_0$  and  $P_0$  is empty.
3.  $Q_0$  is empty, and the condition vacuously holds.
4. Every process  $p_j \in \{p_1, \dots, p_k\} \setminus Q_0 = \{p_1, \dots, p_k\}$  executes  $I'_j$  in  $\alpha_0$ ; there are no nontrivial conflicts in  $\alpha_0$ , and since the STM is  $k$ -progressive every transaction in  $\alpha_0$  has a finite execution interval at the end of which it is logically committed.
5. For every process  $p_j$  from which  $p_0$  does not read,  $\alpha_0^j$  is a  $k$ -independent execution, as all the transactions are nonconflicting, and since the STM is oblivious,  $\alpha_0^j$  is a valid execution. Furthermore,  $\beta_0$  is empty, thus, for every  $j$ ,  $p_j$  does not modify in  $\alpha_0^j$  any base object  $o$ , accessed by  $p_0$ , and the condition trivially holds.

For the induction step, assume an execution  $\alpha_{\ell-1}\beta_{\ell-1}\gamma_{\ell-1}$  satisfies the above conditions. Consider the subset  $V_{\ell-1}$  of  $\{p_1, \dots, p_k\} \setminus P_{\ell-1}$  from which  $p_0$  does not read in  $\alpha_{\ell-1}\beta_{\ell-1}\gamma_{\ell-1}$ . If  $V_{\ell-1}$  is empty then  $p_0$  reads from all the processes and the theorem holds. Otherwise, one of the processes in  $V_{\ell-1}$  is used to construct the next step. For example, for  $\ell = 1$ , Figure 4(b) shows what happens if  $p_i$  is chosen from  $V_0$ , so its execution is perturbed to construct  $\alpha_1\beta_1\gamma_1$ .

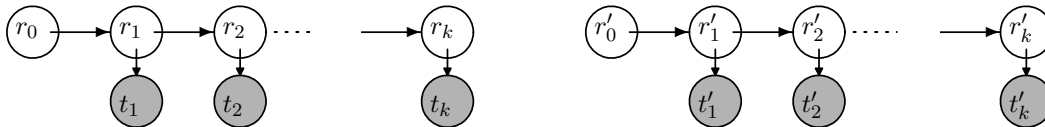
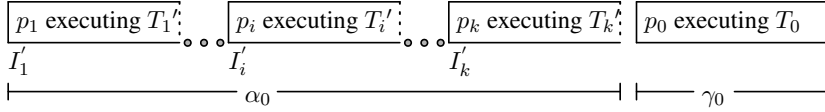
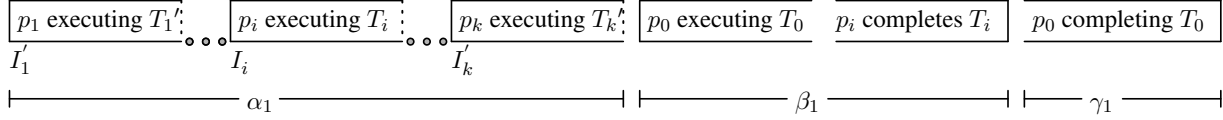


Figure 3: Data setup for the proof of Theorem 4. The privatizing transaction privatizes the left linked-list, while other transactions traverse either the left linked-list or the right linked-list, and write to respective items.



(a) The execution  $\alpha_0\beta_0\gamma_0$ :  $\alpha_0$  is  $I_1^i \dots I_k^i$ ;  $\beta_0$  is the empty execution interval; and  $\gamma_0$  is a solo execution of  $T_0$ .



(b) The execution  $\alpha_1\beta_1\gamma_1$ :  $p_i$  executes  $I_i$  instead of  $I_i^i$  in  $\alpha_1$ ; in  $\beta_1$ ,  $p_0$  starts executing  $T_0$  and  $p_i$  completes  $T_i$ ;  $p_0$  completes  $T_0$  in  $\gamma_1$ .

Figure 4: Executions used in the proof of Theorem 4. A dotted line indicates that the transaction is logically committed.

Pick an arbitrary process  $p_j \in V_{\ell-1}$  and consider the execution  $\alpha_{\ell-1}^j$ , in which  $p_j$  executes  $I_j$  instead of  $I_j^i$ , and other processes take the same steps as in  $\alpha_\ell$ .

The execution  $\alpha_{\ell-1}^j$  is  $k$ -independent, since all the transactions are nonconflicting. Since the STM is oblivious,  $\alpha_{\ell-1}^j$  is a valid execution, and since the STM is  $k$ -progressive  $T_j$  is logically committed in  $\alpha_{\ell-1}^j$ . Since the STM is oblivious,  $\alpha_{\ell-1}^j$  is indistinguishable to every process in  $\{p_1, \dots, p_k\} \setminus \{p_j\}$  from  $\alpha_{\ell-1}$ . Furthermore, by the inductive assumption, that  $p_j$  does not modify in  $\alpha_{\ell-1}^j$  any base object  $o$ , if in  $\alpha_{\ell-1}\beta_{\ell-1}$   $p_0$  reads  $o$  from a process  $p_h$ ,  $h < j$ . Thus,  $p_0$  reads the same values as in the execution of  $\beta_{\ell-1}$ , and there is an execution  $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$  such that  $\beta_{\ell-1}^j$  and  $\beta_{\ell-1}$  are indistinguishable, and  $p_0$  runs solo in  $\gamma_{\ell-1}^j$ .

Assume that  $p_0$  does not read from  $p_j$  also in  $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$ . Then  $p_0$  takes the same steps in  $\gamma_{\ell-1}^j$  and  $\gamma_{\ell-1}$  and  $T_0$  is committed in  $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$ . Let  $m_1, \dots, m_k$  be the base objects that are private to  $p_0$  after  $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$ .

The pending transactions in the execution  $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$  are not conflicting. Since the STM is  $k$ -progressive and  $T_j$  is logically committed after  $I_j$ , it commits (writing to  $t_j$ ) when executed solo after  $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$ . Consider the execution  $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^jJ_j$ , such that  $J_j$  is the execution of  $T_j$  until it commits (see Figure 2(c)). In a manner similar to Lemma 3, we show that  $p_j$  must modify the state of  $m_j$  in  $J_j$  (Lemma 9 in Appendix B). Therefore,  $p_j$  applies a nontrivial primitive to  $m_j$  in some step during  $J_j$ , after the execution of  $T_0$ , in contradiction to privatization safety. Thus,  $p_0$  must read from  $p_j$  in  $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$ .

Let  $s_j$  be the number of steps until  $p_0$  reads from  $p_j$  for the first time in  $\gamma_{\ell-1}^j$ . Pick a process  $p_{j_\ell}$  such that  $s_{j_\ell}$  is the smallest, and if  $s_{j_\ell} = s_{h_\ell}$  then  $j_\ell > h_\ell$ .

Let the execution interval  $\alpha_\ell$  be  $\alpha_{\ell-1}^{j_\ell}$ . The execution interval  $\beta_\ell$  is  $\beta_{\ell-1}$  extended with the first  $s_{j_\ell} - 1$  steps of  $p_0$  in  $\gamma_{\ell-1}^{j_\ell}$ , then a solo execution of  $p_{j_\ell}$  completing  $T_{j_\ell}$ , and finally, the  $s_{j_\ell}$  step of  $p_0$  from  $\gamma_{\ell-1}^{j_\ell}$ , which reads from  $p_{j_\ell}$ . Since  $T_{j_\ell}$  is logically committed in  $\alpha_\ell$ , and the STM is  $k$ -progressive,  $T_{j_\ell}$  commits in  $\beta_\ell$ . The execution interval  $\gamma_\ell$  is defined as a solo execution of  $p_0$  completing  $T_0$ .

It remains to verify the conditions hold for  $\alpha_\ell\beta_\ell\gamma_\ell$ .

1.  $T_0$  completes successfully as there is no incomplete conflicting transaction after  $\alpha_\ell\beta_\ell$ , and the STM is  $k$ -progressive.
2. By the induction assumption,  $p_0$  reads from at least  $\ell - 1$  processes,  $P_{\ell-1}$ , in  $\alpha_{\ell-1}\beta_{\ell-1}$ , not including

$p_{j_\ell}$  that was chosen in the last iteration. The executions  $\alpha_{\ell-1}$  and  $\alpha_\ell$  are indistinguishable to all the processes  $p_h$ , for  $h < j_\ell$ . Furthermore, since the STM is oblivious,  $\alpha_{\ell-1}$  and  $\alpha_\ell$  are indistinguishable to all the processes  $p_h$ , for  $h > j_\ell$ . Hence,  $p_0$  reads from at least the same  $\ell - 1$  processes in  $\alpha_\ell \beta_{\ell-1}$ . In addition,  $p_0$  reads from  $p_{j_\ell}$  in  $\alpha_\ell \beta_\ell$ . Thus,  $P_\ell \supseteq P_{\ell-1} \dot{\cup} \{p_{j_\ell}\}$ , and  $|P_\ell| \geq |P_{\ell-1}| + 1 \geq \ell$ .

3. By the induction assumption,  $Q_{\ell-1}$  is a subset of  $P_{\ell-1}$ , such that every process  $p_h \in Q_{\ell-1}$  executes  $I_h$  in  $\alpha_{\ell-1}$ , and completes  $T_h$  in  $\beta_{\ell-1}$ . Only  $p_{j_\ell}$  is in  $Q_\ell \setminus Q_{\ell-1}$ , and it executes  $I_{j_\ell}$  in  $\alpha_\ell$  and completes  $T_{j_\ell}$  in  $\beta_\ell$ . Since  $\alpha_{\ell-1}$  and  $\alpha_\ell$  are indistinguishable to all the processes in  $\{p_1, \dots, p_k\} \setminus \{p_{j_\ell}\}$ , and since only  $p_{j_\ell}$  switched from  $I'_{j_\ell}$  in  $\alpha_{\ell-1}$  to  $I_{j_\ell}$  in  $\alpha_\ell$ , all the processes  $p_h \in Q_\ell \setminus \{p_{j_\ell}\}$  execute  $I_h$  in  $\alpha_\ell$  and complete  $T_h$  in  $\beta_{\ell-1}$ , which is the prefix of  $\beta_\ell$ .
4. By the induction assumption, every process  $p_h \in \{p_1, \dots, p_k\} \setminus Q_{\ell-1}$ , executes  $I'_h$  in  $\alpha_{\ell-1}$ . Since only  $p_{j_\ell} \in \{p_1, \dots, p_k\} \setminus Q_{\ell-1}$  switched from  $I'_{j_\ell}$  in  $\alpha_{\ell-1}$  to  $I_{j_\ell}$  in  $\alpha_\ell$ , and since  $p_{j_\ell} \notin \{p_1, \dots, p_k\} \setminus Q_\ell$ , every process  $p_h \in \{p_1, \dots, p_k\} \setminus Q_\ell$  executes  $I'_h$  in  $\alpha_\ell$ .
5. Assume, by way of contradiction, that for some  $j$ ,  $p_j$  modifies in  $\alpha_\ell^j$  the base object  $o$ , which in  $\alpha_\ell \beta_\ell$   $p_0$  reads from  $p_h$ ,  $h < j$ . Denote by  $\sigma$  the step during  $\alpha_\ell \beta_\ell$ , in which  $p_0$  reads from  $p_h$ . There is a step,  $\sigma'$ , which is either  $\sigma$  or follows  $\sigma$  during  $\alpha_\ell \beta_\ell$ , in which  $p_0$  reads from  $p_{h_{\ell'}}$   $\in Q_\ell$ , since  $p_0$  always reads from a process in  $Q_\ell$  in the last step of  $\alpha_\ell \beta_\ell$ . Consider the iteration,  $\ell'$ , in which  $p_{h_{\ell'}}$  was chosen to switch from  $T_{h_{\ell'}}$  to  $T_{h_{\ell'}}$ . Since the STM is oblivious, the executions  $\alpha_{\ell'-1}$  and  $\alpha_{\ell'}$  are indistinguishable to the processes in  $\{p_1, \dots, p_k\} \setminus \{p_{h_{\ell'}}\}$ . Process  $p_j$  should have been chosen in the iteration  $\ell'$ , since if  $\sigma'$  follows  $\sigma$ , then  $s_j < s_{h_{\ell'}}$ , otherwise,  $\sigma'$  is  $\sigma$ , i.e.,  $h = h_{\ell'}$  and  $j > h_{\ell'}$ . ■

The proof holds for every workload containing  $m$  items  $t_1, \dots, t_m$  that are privatized by the privatizing transaction, each of which is written by an updating transaction, such that the updating transaction  $T_i$  (writing to  $t_i$ ) does not read any item in  $\{t_1, \dots, t_k\} \setminus \{t_i\}$ . There are no other constraints on the read sets of the transactions: Additional items, like the head node in the linked-list workload, could be privatized and read by all updating transactions; they are just not counted towards the lower bound.

#### 4.1 Reducing the Cost of Privatization by Restricting Parallelism

The previous lower bound, stated as the minimum between the number of privatized items and the level of parallelism, indicates that one way to reduce the cost associated with privatization, is to limit the parallelism offered by the STM. We next show how this tradeoff can be exploited, by sketching a “counter-example” STM, which is a variant of RingSTM [32]. The variant, called *VisibleRingSTM*, reduces the cost of privatization while limiting parallelism.

RingSTM is oblivious and privatization-safe, but not progressive; privatizing  $k$  items accesses  $O(c)$  base objects, where  $c$  is the number of concurrent transactions. RingSTM represents transactions’ read and write sets as Bloom filters [6]. Transactions commit by enqueueing a Bloom filter onto a global ring; the Bloom filter representing the read set of a transaction is only for its internal use. During validation, a transaction  $T$  checks for intersections between the read set of  $T$  and the write sets of other logically committed transactions in the ring, and aborts in case of a conflict. In the commit phase,  $T$  ensures that a write-after-write ordering is preserved. This is done by checking for intersections between the write set of  $T$  and the write sets of other logically committed transactions in the ring. RingSTM is not  $l$ -progressive, for any  $l$ , since a transaction blocks until all concurrent logically committed transactions are completed.

In *VisibleRingSTM*, the read set Bloom filter is visible to other transactions, like the filter of the write set. In the commit phase,  $T$  ensures that a write-after-read ordering is preserved, in addition to the write-

after-write ordering, as in RingSTM. This is done by checking for intersections between the write set of  $T$  and the (visible) read sets of other logically committed transactions in the ring (in addition to checking for intersections with the write sets of these transactions). Intersection between the read set of  $T$  and the write set of another transaction is checked by validation. There is no need to check for intersection between read sets, as these are trivial conflicts that should not interfere. Finally, waiting for all logically committed transactions to complete (at the end of the commit phase) is removed in VisibleRingSTM, as the write-after-read and write-after-write ordering ensure that all the concurrent conflicting transactions have completed.

In VisibleRingSTM, a transaction aborts only due to read-after-write conflicts with other logically committed transactions, and blocks after it is logically committed only due to write-after-write or write-after-read conflicts with other logically committed transactions. A privatizing transaction accesses the  $c$  ring entries of concurrent logically committed transactions, the items in its data set and the global ring index.

The cost of a privatizing transaction can be bounded by  $O(c_0)$ , for any  $c_0 > 1$ , by using a ring of size  $c_0$ ; thus, a privatizing transaction needs to access at most  $c_0$  ring entries. In order to commit, a transaction scans the ring for an empty entry. When there are at most  $c_0$  concurrent transactions, it will find an empty entry, become logically committed, and continue as in VisibleRingSTM. This STM is  $(c_0 - 1)$ -progressive, but a transaction blocks in executions with more than  $c_0$  concurrent transactions (even if they are not conflicting). Thus, the cost of privatization is reduced by limiting the progress of concurrent transactions.

## 5 Bounds for Disjoint-Access Parallel STMs

*Disjoint-access parallelism* [19] is a property that captures the intuition that transactions accessing disjoint parts of the data should not interfere with each other [18].

Somewhat more precisely (cf. [4]), a *conflict graph* represents conflicts between transactions that overlap in time. The vertices in the conflict graph represent transactions; an edge connects two transactions that access the same item. Two transactions  $T_1$  and  $T_2$  are *disjoint-access* if there is no path between them in the conflict graph of the minimal execution interval containing the intervals of  $T_1$  and  $T_2$ . An STM is (*weakly*) *disjoint-access parallel* if two processes executing transactions  $T_1$  and  $T_2$  contend (i.e., have pending primitives, at least one of which is nontrivial, on the same base object at some configuration) only if  $T_1$  and  $T_2$  are not disjoint-access.

With invisible reads, oblivious STMs are a generalization of disjoint-access parallel STMs. More specifically, disjoint-access parallel STMs with invisible reads are oblivious, since nonconflicting transactions do not access the same memory locations, and hence a transaction is always oblivious to different nonconflicting executions. Hence, Theorem 2 implies:

**Corollary 5** *For any privatization-safe STM that is 1-progressive, disjoint-access parallel and with invisible reads, there is a privatization workload in which transactions have nonempty read sets, for which there is an execution where the size of the data set of a transaction privatizing  $k$  items is  $\Omega(k)$ .*

This generalization is strict, since several clock-based STMs [10, 26, 27] are not disjoint-access parallel, but they are oblivious, so our lower bound applies to them, since they have invisible reads.

When reads are visible, we need a notion of disjoint-access parallelism that takes into account only nontrivial conflicts. An edge in a *nontrivial conflict graph* connects two transactions only if they have a nontrivial conflict. An STM is *nontrivial disjoint-access parallel* if two processes executing transactions  $T_1$  and  $T_2$  contend, only if there is a path between  $T_1$  and  $T_2$  in their nontrivial conflict graph.

Nontrivial disjoint-access parallel STMs are oblivious since transactions without nontrivial conflicts do not access the same memory locations. By Theorem 4:

**Corollary 6** *For any privatization-safe STM that is  $l$ -progressive and nontrivial disjoint-access parallel there is a privatization workload in which update transactions have nonempty read sets, for which there is an execution where a transaction privatizing  $m$  items accesses  $\Omega(k)$  base objects, where  $k = \min\{l, m\}$ .*

## 6 Uninstrumented Access without Prior Privatization

*Parameterized opacity* [14] is a framework for describing the interaction between transactions and nontransactional operations, extending *opacity* [15], and parameterized by a memory model for the semantics of nontransactional operations. Roughly, every transaction appears as if it is executed instantaneously with respect to other transactions and nontransactional operations, and nontransactional operations obey the underlying memory model.

Guerraoui et al. [14] prove that for memory models that restrict the order of some pair of read or write operations to different variables, parameterized opacity cannot be achieved [14, Theorem 1]. Furthermore, they prove that for memory models that allow reordering all operations to different variables, parameterized opacity requires either instrumenting nontransactional operations or using RMW primitives when writing inside a transaction. They also present an uninstrumented STM that guarantees parameterized opacity with respect to memory models that do not restrict the order of any pair of read or write operations. This STM uses a global lock, and is not weakly progressive. Their results assume that items are accessed nontransactionally, without a preceding privatization transaction. In a sense, their results show the implications of not privatizing, while our results complete the picture by showing the cost of privatization.

It can be shown that a 1-progressive, oblivious uninstrumented STM, allowing more than one transaction to proceed concurrently, cannot achieve opacity parameterized with respect to any memory model.

**Theorem 7** *There is no uninstrumented, 1-progressive, oblivious STM that guarantees opacity with respect to any memory model.*

The proof (see Appendix C) follows the lines of the proof of Theorem 1 in [14], specifically, the part of the proof that handles memory models that restrict the order of pairs of read operations. Their proof constructs an execution, in which one process executes a transaction and another process issues nontransactional read operations, such that there is no serialization of the transaction and nontransactional operation which respects the restriction posed by the memory model. The memory model in our proof may not pose any restriction on the order of the nontransactional read operations, but we can fix their order by inserting an additional transaction between them.

Together with [14, Theorem 1], this means that progressive, oblivious, uninstrumented STMs cannot achieve opacity parameterized with respect to any memory model, and indicates that a privatizing transaction must precede nontransactional accesses to data items, unless parallelism is compromised.

## 7 Related Work

Many STMs supporting privatization are oblivious [8,9,11,13,24,32] because they avoid the cost of tracking the read sets of other transactions, especially if they are not conflicting. The visibility of reads is not induced by the obliviousness of the STM: Some oblivious STMs use invisible reads [8, 24, 32], making their read set nonexistent for other transactions. Other STMs, e.g., [9], use *partially visible* reads [22], implying that other transactions cannot determine which transaction exactly is reading the item. Some oblivious STMs

even use visible reads, e.g., [13], however, their execution is unaffected by trivial, read-read conflicts. Our lower bounds hold for all these STMs.

TLRW [11] uses read locks, making reads visible. The lock contains a byte per each *slotted* reader, and a reader-count that is modified by other, *unslotted* readers. Slotted readers only write to their slot when reading, so they are unaware of other reads, while unslotted processes read and write to a common counter, and their execution is affected by other reads to the same item (read-read conflict). Therefore, TLRW is oblivious when restricted to slotted readers, and, as predicted by our lower bound, the number of locations accessed by a privatizing transaction is linear in the number of slotted readers.

Our lower bounds indicate that providing efficient privatization requires to compromise parallelism. Inspecting many STMs supporting privatization, e.g., [8,9,13,22,24,25,32], reveals that they limit parallelism, in one way or another.

In *explicit privatization* [30], the application explicitly annotates privatizing transactions, and the STM implementation can be optimized to handle such transactions efficiently; this approach is error-prone and places additional burden on the programmer, which STM tries to avoid in the first place [21]. In *implicit privatization* [22], the STM implementation is required to handle all transactions as if they are potentially privatizing items; this incurs excessive overhead for all transactions.

Some experiments [12, 30, 34] tested techniques used to support implicit privatization in implementations with invisible reads. The results show a significant impact on the scalability and performance relative to STMs supporting explicit privatization; in some cases, the performance degrades to be worse than in sequential code.

*Private transactions* [9] attempt to combine the ease of use of implicit privatization with the efficiency benefits of explicit privatization. A private transaction inserts a *quiescing barrier* that waits till all active transactions have completed; thus other, non-privatizing transactions avoid the overhead of privatization. The barrier accesses an array whose size is proportional to the maximal parallelism, demonstrating again the tradeoff between parallelism and privatization cost, in oblivious STMs.

*Static separation* [2] is a discipline in which each data item is accessed either only transactionally or only nontransactionally. In order to privatize items, the transaction copies them to a private buffer, trivially demonstrating our lower bound. *Dynamic separation* [1] allows data to change access modes without being copied, simply by setting a protection mode in the item. Dynamic separation requires the programmer to access all items to become unprotected, i.e., privatized, as is indicated by our lower bound.

## 8 Discussion

This paper studies the theoretical complexity of privatization that allows uninstrumented nontransactional reads, and shows an inherent cost, linear in the number of privatized items. Privatizing transactions in STMs with invisible reads must have a data set of size  $k$ , where  $k$  is the number of privatized items. A more involved proof shows that even with visible reads, the privatizing transaction must access  $\Omega(k)$  memory locations, where  $k$  is the minimum between the number of privatized items and the number of concurrent transactions that make progress. Both results assume that the STM is oblivious to different non-conflicting executions and guarantees progress in such executions. The specific assumptions needed to prove the bounds indicate that limiting the parallelism or tracking the data sets of other transactions are the price to pay for efficient privatization.

The privatization problem is informally characterized by two subproblems: The *delayed cleanup* problem [20], in which transactional writes interfere with nontransactional operations, and the *doomed transaction* problem [33], in which transactional reads of private data lead to inconsistent state. Our definition of

privatization safety (Property 3) formalizes the first problem; our results show that this problem by itself is an impediment to the efforts to provide efficient privatization.

As discussed in Section 7, some STMs maintain visible reads, yet they are oblivious [9,13]. SkySTM [21] has visible reads, and it avoids the cost of the privatizing transaction by not being oblivious; it makes transactions with trivial read-read conflicts visible to each other. Since SkySTM is not oblivious, our lower bounds do not hold for it. SkySTM, however, demonstrates the alternative cost of not being oblivious, since any writing transaction—not only privatizing transactions—writes to a number of base objects that is linear in the size of its data set, not just the write set. It remains an interesting open question whether this is an inherent tradeoff, or whether there is an STM such that a privatizing transaction accesses  $O(1)$  base objects, and any writing transaction writes to a number of base objects that is linear in the size of its write set.

*Strong privatization* safety [21] further guarantees that no primitive (including a read) is applied to a private location of a process that completed a privatizing transaction. It formalizes the other problem with privatization, of doomed transactions, and it would be interesting to investigate the cost of supporting it.

**Acknowledgements.** We thank Keren Censor Hillel, Panagiota Fatourou, Petr Kuznetsov, Alessia Milani, and Michael Spear for helpful comments.

## References

- [1] M. Abadi, A. Birrell, T. Harris, J. Hsieh, and M. Isard. Implementation and use of transactional memory with dynamic separation. In *CC '09*, pages 63–77.
- [2] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08*, pages 63–74.
- [3] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP '09*, pages 185–196.
- [4] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *SPAA '09*, pages 69–78.
- [5] H. Avni and N. Shavit. Maintaining consistent transactional states without a global clock. In *SIROCCO '08*, pages 131–140.
- [6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
- [7] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008.
- [8] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *PPoPP '10*.
- [9] D. Dice, A. Matveev, and N. Shavit. Implicit privatization using private transactions. In *TRANSACT '10*.
- [10] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC '06*, pages 194–208.
- [11] D. Dice and N. Shavit. TLRW: Return of the read-write lock. In *SPAA '10*.
- [12] A. Dragojevic, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a Research Toy. Technical Report LPD-REPORT-2009-003, EPFL, 2009.
- [13] J. E. Gottschlich, M. Vachharajani, and S. G. Jeremy. An efficient software transactional memory using commit-time invalidation. In *CGO '10*.
- [14] R. Guerraoui, T. Henzinger, M. Kapalka, and V. Singh. Transactions in the jungle. In *SPAA '10*, pages 275–284.

- [15] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP '08*, pages 175–184.
- [16] R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. In *POPL '09*, pages 404–415.
- [17] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS '03*, page 522.
- [18] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
- [19] A. Israeli and L. Rappoport. Disjoint-access parallel implementations of strong shared memory primitives. In *PODC '94*, pages 151–160.
- [20] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.
- [21] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *TRANSACT '09*.
- [22] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *ICPP '08*, pages 67–74.
- [23] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17, 2006.
- [24] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for Java STM. In *SPAA '08*, pages 314–325.
- [25] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *PACT '07*, pages 365–375.
- [26] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC '06*, pages 284–298.
- [27] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *SPAA '07*, pages 221–228.
- [28] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. *SIGPLAN Not.*, 43(10):181–194, 2008.
- [29] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. *SIGPLAN Not.*, 42(6):78–88, 2007.
- [30] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Ordering-based semantics for software transactional memory. In *OPODIS '08*, pages 275–294.
- [31] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. Technical Report Tr 915, Dept. of Computer Science, Univ. of Rochester, 2007.
- [32] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *SPAA '08*, pages 275–284.
- [33] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07*, pages 34–48.
- [34] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *SPAA '08*, pages 265–274.

## A Lemma for Theorem 2

**Lemma 8**  $T_0$  does not access  $t_i$  also when executed after  $I_1$ .

**Proof:** The proof uses terminology that captures the high-level behavior of transactions, which we define here. A transaction includes *access operations*, each operation is a triple: the item which the operation accesses, an indication whether the operation is a read or a write, and the value written or read by the operation. In addition, there are three special operations: the *start*, *commit* and *abort* operations. Every new transaction begins with the start operation, then followed by a sequence of read and write operations. The last operation of a transaction is either an access operation, in which case the transaction is pending, or a commit or abort operation, in which case the transaction is *committed* or *aborted*, respectively.

At any point during the execution of a transaction, the *view* of the transaction is a list of item-value pairs, which includes all the values read by the transaction to this point.

The claim is trivial if that data set is static, so assume that the data set of the transaction is dynamic, i.e., deterministically determined by the view of the transaction before every operation. We prove by induction that the transaction has the same view after applying the same number of operations when executed after  $I_1$  and  $I'_1$ , and that  $T_0$  logically commit when running after  $I_1$ .

The base case is before applying the first operation: the view of the transaction is empty in both cases, and  $T_0$  is neither aborted nor blocked.

For the induction step, assume that after applying  $i - 1$  operations when running after  $I_1$ ,  $T_0$  has the same view as after applying  $i - 1$  operations when running after  $I'_1$ , and that  $T_0$  is neither aborted nor blocked. The  $i$ -th operation  $T_0$  applies when running after  $I_1$ , is the same as the  $i$ -th operation  $T_0$  applies when running after  $I'_1$ . Recall that  $T_0$  does not access  $u$  in any execution. Since  $T_0$  does not access  $t_i$  when executed after  $I'_1$ , the  $i$ -th operation  $T_0$  applies when running after  $I_1$  also does not access  $t_i$ .  $T_0$  does not abort nor blocks when applying the  $i$ -th operation after  $I_1$ , and the view after the  $i$ -th operation is the same view as after applying the  $i$ -th operation when running after  $I'_1$ . ■

## B Lemma for Theorem 4

**Lemma 9**  $p_j$  modifies the state of  $m_j$  in  $J_j$ .

**Proof:** We first show that  $p_j$  does not apply nontrivial primitive to  $m_j$  in  $\alpha_{\ell-1}^j$ . Otherwise, let  $\tau$  be the first such step. Let  $\hat{\alpha}_{\ell-1}^j$  be the prefix of  $\alpha_{\ell-1}^j$  preceding  $\tau$ . Consider the execution  $\hat{\alpha}_{\ell-1}^j\beta$ , where each pending transaction preceding  $I_j$  in  $\alpha_{\ell-1}^j$  executes solo to completion in  $\beta$ ; since there are no nontrivial conflicts in  $\hat{\alpha}_{\ell-1}^j\beta$ , and the STM is  $k$ -progressive, all the transactions running in  $\beta$  complete. Let  $C$  be the configuration at the end of this execution. Only  $T_j$  is pending in  $C$ , however,  $T_j$  is not logically committed in  $C$ . In a solo execution of  $T_0$  from  $C$ ,  $T_0$  is committed, making  $m_j$  private to  $p_0$ . Since the STM is not eager,  $p_j$  does not apply  $\tau$  in  $C$ .

A similar argument shows that  $p_j$  does not apply nontrivial primitive to  $m_j$  also in  $\alpha_{\ell-1}$ .

Next, we prove that  $p_0$  does not modify the state of  $m_j$  in  $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$ . Otherwise, a nontransactional read operation to  $t_j$  of  $p_0$  after  $\alpha_{\ell-1}\beta_{\ell-1}\gamma_{\ell-1}$  returns a value that is not the initial value of  $m_j$ , whereas no committed transaction writes to  $t_j$  in  $\alpha_{\ell-1}\beta_{\ell-1}\gamma_{\ell-1}$ , contradicting our correctness condition. Since  $p_0$  does not read from  $p_j$  also in  $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$ ,  $p_0$  applies the same steps in  $\beta_{\ell-1}^j\gamma_{\ell-1}^j$  and in  $\beta_{\ell-1}\gamma_{\ell-1}$ . As we have shown,  $p_j$  does not apply a nontrivial primitive to  $m_j$  neither in  $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$  nor in  $\alpha_{\ell-1}\beta_{\ell-1}\gamma_{\ell-1}$ , thus,  $p_0$  does not modify the state of  $m_j$  also in  $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j$ .

If  $p_j$  does not modify the state of  $m_j$  also in  $J_j$ , then a nontransactional read by  $p_0$  to  $t_j$  after  $\alpha_{\ell-1}^j \beta_{\ell-1}^j \gamma_{\ell-1}^j J_j$  returns the initial value of  $m_j$ , since reads are uninstrumented. This contradicts our correctness condition since  $T_j$ , which writes to  $t_j$  a value that is different from the initial value of  $m_j$ , commits before the nontransactional read by  $p_0$ . ■

## C Proof of Theorem 7

**Theorem 7.** *There is no uninstrumented, 1-progressive, oblivious STM that guarantees opacity with respect to any memory model.*

**Proof:** Consider an execution of a transaction  $T_1$  by  $p_1$ , writing  $v_1$  to item  $x$  and  $v_2$  to item  $y$ . Denote by  $m_x^2, m_y^2$  the base objects that  $p_2$  accesses when it reads nontransactionally from  $x$  and  $y$  respectively.

During the execution of  $T_1$ ,  $p_1$  writes  $v_1$  to  $m_x^2$ , and  $v_2$  to  $m_y^2$ . Otherwise, since the STM is uninstrumented, a nontransactional read of  $p_2$  from  $x$  (respectively,  $y$ ) after  $T_1$  is completed, returns the initial value of  $m_x^2$  (respectively,  $m_y^2$ ) instead of  $v_1$  (respectively,  $v_2$ ), so the STM is not parameterized opaque, and we are done.

Without loss of generality, assume  $p_1$  writes  $v_1$  to  $m_x^2$  before writing  $v_2$  to  $m_y^2$ . Denote by  $C_1$  the configuration after  $p_1$  first writes  $v_1$  to  $m_x^2$ . Consider the solo execution of  $p_2$  from  $C_1$  in which it reads  $x$  nontransactionally, executes a transaction  $T_2$  in which it writes to item  $z$ , and then reads  $y$  nontransactionally.  $T_2$  is committed since the STM is 1-progressive. Denote by  $C_2$  the configuration after the solo execution of  $p_2$ , and by  $\alpha$  the execution of  $p_1$  and  $p_2$  preceding  $C_2$ . Process  $p_1$  completes  $T_1$  from  $C_2$ . The nontransactional read of  $x$  by  $p_2$  returns  $v_1$ , and parameterized opacity implies that  $T_1$  is logically committed at  $C_1$ , since the STM is 1-progressive  $T_1$  commits when executed from  $C_2$ .

**Claim 10** *Process  $p_2$  does not modify the state of  $m_y^2$  in  $\alpha$ .*

**Proof:** Consider an execution  $\alpha'$ , such that  $p_1$  executes solo a transaction  $T_1'$  writing to item  $w$  until the transaction is logically committed at configuration  $C_1'$ . Then,  $p_2$  runs solo reading  $x$  nontransactionally, executing  $T_2$ , and reading  $y$  nontransactionally. Since the STM is uninstrumented  $p_2$  does not modify the state of  $m_y^2$  when reading nontransactionally  $x$  and  $y$ .

If  $p_1$  or  $p_2$  modify the state of  $m_y^2$  while executing  $T_1'$  and  $T_2$  in  $\alpha'$  then the nontransactional read operation to  $y$  of  $p_2$  after  $T_2$  returns a value that is not the initial value of  $m_y^2$ , whereas no committed transaction writes to  $y$  in  $\alpha'$ , contradicting parameterized opacity.

By 1-obliviousness, the executions of  $T_2$  from  $C_1$  and  $C_1'$  are indistinguishable. Process  $p_1$  does not access  $m_y^2$  neither in  $\alpha$  nor in  $\alpha'$ , thus  $p_2$  does not apply successfully a nontrivial primitive to  $m_y^2$  also in  $\alpha$ . ■

Parameterized opacity implies that  $T_1$  appears before the nontransactional read to  $x$  in the ordering, and that  $T_2$  and the nontransactional read to  $y$  are ordered after the nontransactional read to  $x$ , and thus after  $T_1$ . However, the nontransactional read to  $y$  is issued before  $p_1$  first writes to  $m_y^2$  and by Claim 10 also  $p_2$  does not modify the state of  $m_y^2$ . Since the STM is uninstrumented, the nontransactional read to  $y$  returns the initial value of  $m_y^2$  and not  $v_2$ , thus the STM is not parameterized opaque. ■