The Cost of Privatization⋆

Hagit Attiya[1,2] and Eshcar Hillel[1]

[1] Department of Computer Science, Technion
[2] Ecole Polytechnique Federale de Lausanne (EPFL)

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* [19, 22, 28] 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, 27], they do not alleviate it completely. Thus, STMs seek to improve performance by supporting *uninstrumented* nontransactional operations [14, 30], which are executed as is, typically as a single access to the shared memory.

Many recent STMs [8, 9, 11, 13, 20, 21, 23, 24, 31] provide strong atomicity by supporting *privatization* [28, 30], 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, 20, 31], e.g., by nullifying these references. However, it has been claimed that privatization is a major source of overhead for transactional memories [33], 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 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, 24], 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.

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
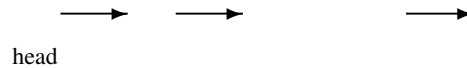

head

Fig. 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.

clock or counter [10, 25, 26], 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 [8,9,13,23,31] are also oblivious. (A detailed discussion appears in Section 5.)

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 1 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 2 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, 23, 30], for example, must compromise parallelism in order to support efficient privatization.

Obliviousness generalizes *disjoint-access parallelism* [18], and our lower bounds hold also for disjoint-access parallel STMs (see [4]).

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.

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 α_1 and α_2 are *indistinguishable* to a process p, if p goes through the same sequence of state changes in α_1 and in α_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] 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[3] 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 α 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 $k \geq 1$) 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 $\ell \geq 0$ transactions, each executed by a different process, $p_{i_1}, \ldots, p_{i_\ell}$, 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 α_1 and α_2, each containing ℓ transactions executed by the same processes $p_{i_1}, \ldots, p_{i_\ell}$, in the same order, if some transaction T executed by a process p does not have nontrivial conflicts with the transactions in α_1 and α_2, then $\alpha_1 T$ and $\alpha_2 T$ are indistinguishable to p.

---

[3] A *conflict* occurs when two operations access the same data item; the conflict is *nontrivial* if one of the operations is a write.

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 α 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 α' that includes T' instead of T, such that α' is indistinguishable to all other processes from α.

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 [20] 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 (see [4]). Note that assuming uninstrumented nontransactional operations implies that the mapping of each privatized item to the corresponding base object is static. In the sequel, we assume that a privatization-safe STM is not eager.

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 1. *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 $t_1, \ldots t_k$. 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 (see [4]) 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 α. Let $m_1, \ldots, m_k$ be the base objects that are private to p_0 after α (we omit the superscript 0). Since the STM is 1-progressive T_1 commits when completed after α. 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 1. 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 α. Assume that p_1 applies a nontrivial primitive to m_i in some step when executing I_1 in α, and let τ be the first such step. Let $\widehat{I_1}$ be the prefix of I_1 preceding τ. 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 τ after $\widehat{I_1}$.

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

Next, we argue that p_0 does not modify the state of m_i in α. Otherwise, a nontransactional, uninstrumented read operation, to t_i of p_0 after α' returns a value that is not the initial value of m_i, whereas no committed transaction writes to t_i in α', contradicting our correctness condition. The executions of T_0 after I_1 and I_1' are indistinguishable,

p_1 executing $T_1^{'}$ p_0 executing T_0

$I_1^{'}$

(a) The execution α'.

p_1 executing T_1 p_0 executing T_0

I_1

(b) The execution α.

p_1 executing T_1 p_0 executing T_0 p_1 completes T_1

I_1 J_1

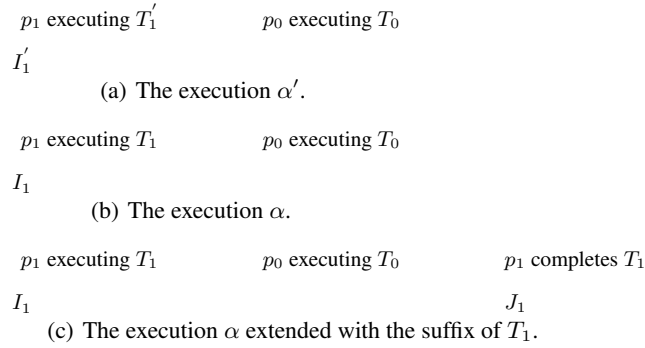(c) The execution α extended with the suffix of T_1.

Fig. 2. The executions used in the proof of Theorem 1. A dotted line indicates that the transaction is logically committed.

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 α nor in α', and hence, p_0 does not successfully apply a nontrivial primitive to m_i also in α.

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 $I_1 T_0 J_1$ 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 1; however, the technical details are more involved, in order to handle visible reads.

Theorem 2. *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 outline it for the linked-list workload of Figure 1; a detailed proof sketch appears in Appendix A.

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; see Appendix A.)

Reducing the cost of privatization by restricting parallelism: The 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 [31]. 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 enqueuing 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 Related Work

Many STMs supporting privatization are oblivious [8, 9, 11, 13, 23, 31] 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, 23, 31], making their read set nonexistent

for other transactions. Other STMs, e.g., [9], use *partially visible* reads [21], 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, 21, 23, 24, 31], reveals that they limit parallelism, in one way or another.

In *explicit privatization* [29], 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 [20]. In *implicit privatization* [21], 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, 29, 33] 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.

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. Guerraoui et al. [14, Theorem 1] prove that parameterized opacity cannot be achieved for memory models that restrict the order of some pair of read or write operations to different variables. They also show that if operations on different variables can be reordered, then parameterized opacity with uninstrumented operations requires RMW primitives when writing inside a transaction. Their results assume that items are accessed nontransactionally, without a preceding privatization transaction. These results indicate the *implications of not privatizing*, while our results show *the cost of privatization*.

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; it uses a global lock, and is not weakly progressive. In the full version of this paper [4], we show that a 1-progressive, oblivious uninstrumented STM, cannot achieve opacity parameterized with respect to *any* memory model. This indicates that a privatizing transaction must precede nontransactional accesses to data items, unless parallelism is compromised.

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.

6 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 [19], in which transactional writes interfere with nontransactional operations, and the *doomed transaction* problem [32], 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 5, some STMs maintain visible reads, yet they are oblivious [9, 13]. SkySTM [20] 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 [20] 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.

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 and E. Hillel. The cost of privatization. Technical Report CS-2010-11, Department of Computer Science, Technion, 2010.
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. A. Israeli and L. Rappoport. Disjoint-access parallel implementations of strong shared memory primitives. In *PODC '94*, pages 151–160.
19. J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.
20. Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *TRANSACT '09*.
21. 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.
22. M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17, 2006.
23. 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.
24. M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *PACT '07*, pages 365–375.
25. T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC '06*, pages 284–298.

26. T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *SPAA '07*, pages 221–228.
27. 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.
28. 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.
29. 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.
30. 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.
31. M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *SPAA '08*, pages 275–284.
32. 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.
33. 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   More Details for the Proof of Theorem 2

We sketch the proof for a workload similar to the linked list of Figure 1; after the proof, we discuss how it can be extended to other scenarios.

Theorem 2. *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 $k = \min\{l, m\}$.*

Sketch of proof. Consider the scenario described in Figure 1: Let $r_0, r_1, \ldots, r_k$ be the nodes of the linked list (with r_0 being 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 $p_0, \ldots, p_k$. Process p_0 executes a transaction T_0 that privatizes the linked list, including the items $t_1, \ldots t_k$. For every process p_i, $1 \leq i \leq k$, we consider 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 $\{r_0, r_1, \ldots, r_i\}$, and its write set is $\{t_i\}$.

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

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

Consider the following execution $\alpha_0 \beta_0 \gamma_0$: α_0 is $I'_1 \ldots I'_k$, such that p_i executes in I'_i, a prefix of the transaction T_i' and T_i' is logically committed after I'_i; β_0 is the empty execution interval; and γ_0 is a solo execution of T_0 by p_0 to completion.

For every ℓ, $0 < \ell \leq k$, 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_ℓ, a subset of $\{p_1,\ldots,p_k\}$ of size (at least) ℓ, in $\alpha_\ell\beta_\ell$.
3. There is a subset Q_ℓ of P_ℓ, where every process $p_j \in Q_\ell$ executes I_j, a prefix of the transaction T_j, in α_ℓ, such that T_j is logically committed after I_j, and p_j completes T_j in β_ℓ.
4. Every process p_j, $\{p_1,\ldots,p_k\} \setminus Q_\ell$, executes I'_j in α_ℓ.
5. For every process p_j from which p_0 does not read in $\alpha_\ell\beta_\ell\gamma_\ell$, α_ℓ^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 α_ℓ; in α_ℓ^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 ℓ. The base case is straightforward (see [4]). 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 $\{p_1,\ldots,p_k\} \setminus P_{\ell-1}$ 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.

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, and other processes take the same steps as in α_ℓ.

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 $\{p_1,\ldots,p_k\} \setminus \{p_j\}$ from $\alpha_{\ell-1}$. Furthermore, the inductive assumption implies 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 $m_1,\ldots,m_k$ 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 $\alpha_{\ell-1}^j\beta_{\ell-1}^j\gamma_{\ell-1}^j J_j$, such that J_j is the execution of T_j until it commits. In a manner similar to Lemma 1, we show that p_j must modify the state of m_j in J_j (see [4]). Therefore, p_j applies a nontrivial primitive to m_j in some step during J_j, which is pending 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_ℓ} such that s_{j_ℓ} is the smallest, and if $s_{j_\ell} = s_{h_\ell}$ then $j_\ell > h_\ell$.

Let the execution interval α_ℓ be $\alpha_{\ell-1}^{j_\ell}$. The execution interval β_ℓ 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_ℓ} completing T_{j_ℓ}, and finally, the s_{j_ℓ} step of p_0 from $\gamma_{\ell-1}^{j_\ell}$, which reads from p_{j_ℓ}. Since T_{j_ℓ} is logically committed in α_ℓ, and the STM is k-progressive, T_{j_ℓ} commits in β_ℓ. The execution interval γ_ℓ 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_ℓ} that was chosen in the last iteration. The executions $\alpha_{\ell-1}$ and α_ℓ are indistinguishable to all the processes p_h, for $h < j_\ell$. Furthermore, since the STM is oblivious, $\alpha_{\ell-1}$ and α_ℓ 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_ℓ} 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_ℓ} is in $Q_\ell \setminus Q_{\ell-1}$, and it executes I_{j_ℓ} in α_ℓ and completes T_{j_ℓ} in β_ℓ. Since $\alpha_{\ell-1}$ and α_ℓ are indistinguishable to all the processes in $\{p_1, \ldots, p_k\} \setminus \{p_{j_\ell}\}$, and since only p_{j_ℓ} switched from I'_{j_ℓ} in $\alpha_{\ell-1}$ to I_{j_ℓ} in α_ℓ, all the processes $p_h \in Q_\ell \setminus \{p_{j_\ell}\}$ execute I_h in α_ℓ and complete T_h in $\beta_{\ell-1}$, which is the prefix of β_ℓ.
4. By the induction assumption, every process $p_h \in \{p_1, \ldots, p_k\} \setminus Q_{\ell-1}$, executes I'_h in $\alpha_{\ell-1}$. Since only $p_{j_\ell} \in \{p_1, \ldots, p_k\} \setminus Q_{\ell-1}$ switched from I'_{j_ℓ} in $\alpha_{\ell-1}$ to I_{j_ℓ} in α_ℓ, and since $p_{j_\ell} \notin \{p_1, \ldots, p_k\} \setminus Q_\ell$, every process $p_h \in \{p_1, \ldots, p_k\} \setminus Q_\ell$ executes I'_h in α_ℓ.
5. Assume, by way of contradiction, that for some j, p_j modifies in α_ℓ^j the base object o, which in $\alpha_\ell \beta_\ell$ p_0 reads from p_h, $h < j$. Denote by σ the step during $\alpha_\ell \beta_\ell$, in which p_0 reads from p_h. There is a step, σ', which is either σ or follows σ 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_ℓ in the last step of $\alpha_\ell \beta_\ell$. Consider the iteration, ℓ', 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 $\{p_1, \ldots, p_k\} \setminus \{p_{h_{\ell'}}\}$. Process p_j should have been chosen in the iteration ℓ', since if σ' follows σ, then $s_j < s_{h_{\ell'}}$, otherwise, σ' is σ, i.e., $h = h_{\ell'}$ and $j > h_{\ell'}$. ∎

The proof holds for every workload in which the updating transaction T_i does not read any item from $\{t_1, \ldots t_k\} \setminus \{t_i\}$.