The Cost of Privatization in Software Transactional Memory

Hagit Attiya , Eshcar Hillel

Abstract—*Software transactional memory* (STM) is a promising approach for programming concurrent applications; STM guarantees that a *transaction*, consisting of a sequence of operations on the memory, appears to execute atomically. In practice, however, it is important to be able to run transactions together with *nontransactional* legacy code accessing the same memory locations, by supporting *privatization* of shared data. 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 r memory locations must be accessed by a privatizing transaction, where r is the minimum between k, the number of privatized items, and the number of concurrent transactions guaranteed to make progress. This captures, in a concrete and quantitative manner, the tradeoff between the cost of privatization and the level of parallelism offered by the STM.

Index Terms—transactional memory, disjoint-access parallelism, privatization safety, progress properties

1 Introduction

Software transactional memory (STM) is an attractive paradigm for programming parallel applications for multicore systems. Inspired by classical database transactions [36], general-purpose 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 on a set of *data items* and guarantees that if any operation takes place, they all do, and that if they do, they appear to do so atomically, as one indivisible computation.

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 [22], [25], [31] 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], [30], they do not alleviate it completely. Thus, STMs aim to support *uninstrumented* nontransactional operations [15], [33], which are executed as is, typically as a single access to the shared memory.

Many recent STMs [8], [9], [11], [14], [23], [24], [26], [27], [34] provide strong atomicity by supporting *privatization* [31], [33], thereby allowing to "isolate" some items and make them private to a single process; the process can thereafter access them nontransactionally by simple memory accesses, without interference by other processes.

Ideally, privatizing a set of items would simply involve disabling all shared references to the items [9], [23], [34], e.g., by nullifying these references. Consider, for example, the linked-list of Figure 1, in which every node points to the root item to a disjoint subgraph, such as a tree, and is the only path to items in the subgraph. A process privatizes all root items and their subgraphs, so it can afterwards access all items in the rooted trees with simple (uninstrumented) reads and writes to the shared memory, avoiding the overhead of transactions and thereby dramatically increasing the efficiency.

What happens, however, when another transaction reads all the nodes of the linked list but writes only to one root item that is pointed from the list? It has been suggested that it suffices for the privatizing transaction to conflict with the other transaction accessing the privatized region; for example, the privatizing transaction writes to the head of the linked list, and any updating transaction reads the head.

This paper shows, under assumptions that hold for

- *H. Attiya is with the Technion, hagit@cs.technion.ac.il*
- *E. Hillel is with the Technion and Yahoo! Labs, eshcar@yahoo-inc.com*
- *A preliminary version of this paper appeared in the* International Symposium on Distributed Computing *(DISC), 2010.*
- *This research is supported in part by the* Israel Science Foundation *(grants 1344/06 and 1227/10), and by funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 238639, ITN project TRANSFORM.*

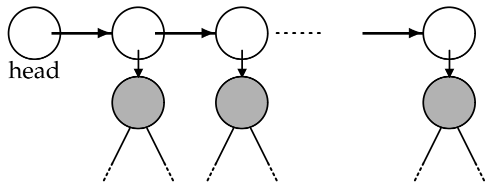


Fig. 1. In this example, the privatizing transaction sets the head to NULL and privatizes the dark items and their subgraphs. Other transactions write to the dark items; each transaction writes to a different one.

many STMs, that a constant amount of work, e.g., nullifying the head of the linked list, does not suffice for privatizing the whole linked list. For many important *workloads*, including the linked-list example, the hope to combine efficient privatizing transactions with uninstrumented nontransactional reads, cannot be realized unless transactional reads are visible or the STM is overly sequential. 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. This holds even if the read set of writing transactions is arbitrary large.[1]

Our lower bounds do not hold for 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], [27], thereby significantly reducing the throughput. Our progress assumption (Property 1, defined in Section 2), requires the STM to allow concurrent progress of nonconflicting transactions: a transaction can abort or block only due to a conflicting pending transaction. This condition holds for *weakly progressive* [17] and *obstruction-free* [18] STMs.

Our lower bounds only assume that after a transaction privatizes a data item, no other transaction can write to it (Property 4 in Section 2.2). This property is common to all interpretations of privatization, e.g., it follows from strong atomicity, implying that our lower bounds hold for all of them.

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, defined in Section 2). A simple example is provided by STMs using a global clock or counter [10], [28], [29], or a decentralized clock [5]: The clock or the counter increases whenever a transaction T writes to some data item; however, another transaction T' cannot tell whether T wrote to item x or item y, unless T' has an operation on either x or y. A less-obvious example is the behavior of TLRW [11] for so-called *slotted* threads (see Section 2.1). Several other STMs [8], [9], [14], [26], [34] are also oblivious. (See detailed discussion in Section 6.)

Our first main result (Theorem 2 in Section 3) 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 $\Omega(k)$. In an oblivious STM with invisible reads, transactions are unaware of, and hence, unaffected by, trivial 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 (Theorem 4 in Section 4) removes the assumption of invisible reads. It shows an $\Omega(r)$ lower bound on the number of shared memory accesses performed by a privatizing transaction, where r is the minimum between k, the number of privatized items, and the level of parallelism, i.e., the number of transactions guaranteed to make progress concurrently. This lower bound captures the tradeoff between the cost of privatization and the parallelism offered by the STM—it explains why the *quiescence* mechanism [9], [26], [33], for example, compromises parallelism in order to support efficient privatization.

We sketch *VisibleRingSTM*, demonstrating how the cost of privatization can be reduced by limiting the parallelism offered by the STM. This shows that our lower bound on the tradeoff between the cost of privatization and the level of parallelism is tight (Section 4.3).

Our results are aligned with observations that privatization leads to an unavoidable overhead for STMs [37], and that efficient support for uninstrumented nontransactional operations comes at the cost of reduced parallelism [7].

Section 5 explains how obliviousness generalizes *disjoint-access parallelism* [21], and derives the lower bounds also for disjoint-access parallel STMs.

The rest of the paper is organized as follows: we start in Section 2, with basic definitions and description of various STM properties needed for our results; we show that *eager* STMs, in which a transaction may update the memory before it is guaranteed to commit cannot support privatization (Theorem 1 in Section 2.2). 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 outline an STM that matches the lower bound. Section 5 presents the results for disjoint-access parallel STMs. In Section 6, we review in detail related work, and in particular, the cost incurred by STMs supporting privatization. Section 7 conclude with a discussion of the results and directions for further work.

1. It can be easily shown that *if the read set of every writing transaction is empty*, then the privatizing transaction must have operations on all the items it privatizes. Otherwise, a transaction writing to this item executed after the privatizing transaction completes, is unaware of the privatization and may access private locations. (This relies on a weak progress assumption.)

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.

An *operation* is a triple: an item, 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: *start*, *commit* and *abort*. A transaction begins with the start operation, followed by a sequence of read and write operations. If the last operation of a transaction is commit or abort, then the transaction is *complete*, and said to be *committed* or *aborted*, respectively; otherwise, the transaction is *pending*.

The collection of data items in the operations of 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 *writing transaction* is a transaction with a nonempty write set.

Two operations *conflict* if they are on the same data item; the conflict is *trivial* if both are read operations, and *nontrivial* otherwise. Two transactions *conflict* if they include conflicting operations; the conflict is trivial or nontrivial, depending on the type of conflict between the operations.

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, like CAS and kCAS. 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 one or more 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 base object. In the unique *initial* configuration, every process is in its initial state and every base object contains its initial value.

An *execution interval* α is a finite or infinite alternating sequence $C_0, \phi_0, C_1, \phi_1, C_2, \ldots$, where C_k is a configuration, ϕ_k is an event and the application of ϕ_k to C_k results in C_{k+1}, for every $k = 0, 1, \ldots$. An *execution* is an execution interval in which C_0 is the initial configuration.

A *solo execution* of process p is an execution interval in which all steps are taken by process p. A transaction *eventually executes solo* if a suffix of the transaction is a solo execution.

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, p goes through the same sequence of events.

2.1 STM Properties

A transaction T is *logically committed* in a configuration C if T does not abort in any infinite extension from C.

Our progress condition requires a transaction to commit if it has no nontrivial conflict with a pending transaction; that is, a transaction can abort or block (take an infinite number of steps without completing) only due to a nontrivial conflict with another transaction. The next property captures the degree of parallelism provided by an STM, in the absence of nontrivial conflicts.

Property 1 (l-progressive STM): An STM is *l-progressive*, $l \geq 0$, if a transaction T that eventually executes solo, after an execution α containing $\leq l$ pending transactions, commits unless it has nontrivial conflict with a pending 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 (for any $l \geq 1$) is satisfied by *weakly progressive* STMs [17], in which a transaction must commit if it does not encounter nontrivial conflicts, and by *obstruction-free* STMs [18], in which a transaction commits when it runs by itself for long enough, implying that it must commit if it runs solo after an execution without nontrivial conflicts.

To increase efficiency, it is desirable that transactions do not observe operations of other transactions without nontrivial conflicts, whether or not these operations leave a mark on the memory. This is captured by the notion of obliviousness, which implies that a solo execution of one transaction can be replaced with a solo execution of another transaction, but other transactions without nontrivial conflicts cannot observe this change.

The formal definition (Property 2) generalizes this concept to a *sequence* of solo executions of several transactions. An *independent* execution contains transactions on data sets without nontrivial conflicts, each executed by a different process, running solo until it is logically committed. Intuitively, an STM is *oblivious* if a transaction running solo after an independent execution of transactions, 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): Let α_1 and α_2 be a pair of independent executions with the same number of writing transactions. Let p be a process that has no steps in α_1 and α_2, and let T be a transaction by p, without nontrivial conflicts with neither the transactions in α_1 nor the transactions in α_2. An STM is *oblivious* if $\alpha_1 T$ 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 without any read operations. More formally:

Property 3 (STM with invisible reads): 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. An STM has *invisible reads* if there is an execution α' that includes T' instead of T, such that α' is indistinguishable to all other processes from α.

Example 1: We demonstrate the definitions by considering TLRW [11]. This is a lock-based STM, in which each item has an associated lock, containing a byte for 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, making TLRW oblivious when restricted to slotted readers. On the other hand, unslotted threads read and write to the reader-count, and their execution is affected by other reads to the same item (trivial, read-read conflict), making TLRW non-oblivious when taking into account non-slotted readers.

To acquire a lock for writing, thread i sets the owner field of the item's lock from $\perp$ to i, and then spins until all readers have completed, i.e., all the bytes (slots) and the reader count are 0. To acquire a lock for reading, a slotted thread stores 1 in its slot, and verifies that the owner field is $\perp$. If the owner field is non-$\perp$, the reader stores 0 into its slot, and repeats until no owner is detected. Unslotted readers follow a similar procedure, but instead of setting a slot they increment and decrement the reader count. Clearly, TLRW has visible reads.

A transaction aborts only due to a timeout by the thread, while it is trying to acquire a lock; therefore, there is a nontrivial conflict implying that TLRW is n-progressive. However, since a transaction acquires locks as the items are accessed, committing only involves writing the new values to the items, before releasing the locks.

Finally, note that a transaction is logically committed after it has acquired the last lock, since it will not abort later; nevertheless, the transaction is not committed yet, and still has to apply its writes.

2.2 Privatization

Instead of attempting to formally define privatization, we only state a property that is naturally expected out of any notion of privatization.

We assume a process p_j can execute a transaction T that *privatizes* a set of items $\{t_1, \ldots, t_k\}$. After the privatizing transaction commits, we assume that p_j owns $t_1, \ldots, t_k$, and say that either process p_j or transaction T *privatizes* them; intuitively, this means that no other process can write to them. Let m_i^j be the private base object that process p_j associates with a data item t_i, after privatizing it; the formal requirement is:

Property 4 (Privatization-safe STM): An STM 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.

Assume that outside a transaction, read operations to a privatized data item are *uninstrumented*, namely, they are simply a primitive read of the private base object associated with the data item.

We assume a weak safety property that only requires a nontransactional read of a data item, without a preceding nontransactional write, to return the value written by an earlier committed transaction, or the initial value, if no such transaction commits. No additional consistency property is used in our proofs.

We now consider whether a privatization-safe STM can be eager, that is, have transactions that apply a nontrivial primitive (e.g., a write) to a base object associated with a privatized item, before it is logically committed. For the purpose of this paper, we say that an STM is *eager* if there is a configuration C such that in C, a transaction T executed by process p is not logically committed, and p applies a nontrivial primitive to a base object associated with an item that another process privatizes.

The following simple theorem shows that a 1-progressive eager STM is not privatization safe. This is claimed to be a known flaw in eager STMs that are not strongly atomic [13], but was never proved formally, and Proposition 1 highlights the assumptions needed for proving it. Note that assuming uninstrumented nontransactional reads implies that the mapping of each privatized item to the corresponding base object is independent of the execution, and is known in advance.

Proposition 1: An uninstrumented 1-progressive eager STM is not privatization safe.

Proof: Let m_i^0 be the private base object which process p_0 associates with an item t_i. Assume another process p_1 executing a transaction T_1 applies a nontrivial primitive to m_i^0 in some configuration C, where only T_1 is pending and before it is logically committed; call this event τ. Consider a transaction T_0 of p_0 privatizing the item t_i, executed from C.

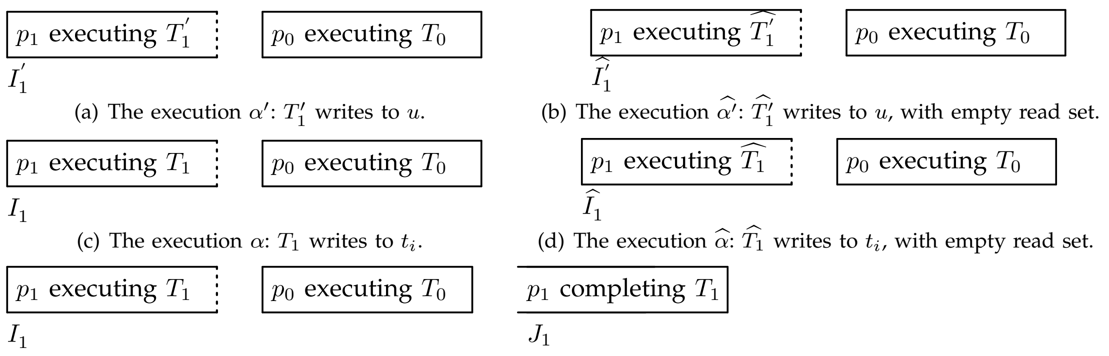
Since only T_1 is pending in C, α, the execution leading to C, has no pending logically committed transaction that has a nontrivial conflict with T_0. Since the STM is 1-progressive, T_0 completes successfully when executed after α, and since the STM is uninstrumented, m_i^0 is private to p_0 after αT_0. However, τ can be applied to m_i^0, even after T_0, in contradiction to privatization safety. $\square$

Below, we assume that the implementations are uninstrumented and (at least) 1-progressive, therefore, Proposition 1 holds, and a privatization-safe STM is not eager.

3 Privatization with Invisible Reads

We show that when a privatization-safe STM is oblivious and has invisible reads, 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 workload including a transaction T_0, privatizing k items, in which transactions have nonempty read sets, for which there is an execution where the size of the data set of T_0 is $\Omega(k)$.



(a) The execution α': T_1' writes to u.

(b) The execution $\widehat{\alpha}'$: $\widehat{T}_1'$ writes to u, with empty read set.

(c) The execution α: T_1 writes to t_i.

(d) The execution $\widehat{\alpha}$: $\widehat{T_1}$ writes to t_i, with empty read set.

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

Fig. 2. The executions used in the proof of Theorem 2. Invisible reads property is used to show that the executions 2(a) and 2(b), and 2(c) and 2(d) are indistinguishable; Obliviousness is used to show that the executions 2(b) and 2(d) are indistinguishable. A dotted line indicates that the transaction is logically committed.

Proof: Consider a workload in which process p_0 executes a transaction T_0 that privatizes the items $t_1, \dots t_k$, while process p_1 executes a transaction T_1' writing u, an unrelated data item,[2] and reading an arbitrary set.

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 (Figure 2(a)), and then p_0 executes T_0. Assume by way of contradiction that the data set of T_0 in α' does not include some privatized item t_i.

Since reads are invisible, α' is indistinguishable to p_0 from an execution $\widehat{\alpha}' = \widehat{I_1'} T_0$, in which p_1 executes a prefix $\widehat{I_1'}$ of a transaction $\widehat{T_1'}$ with an empty read set that writes to u the same value as T_1', until $\widehat{T_1'}$ is logically committed (Figure 2(b)), followed by T_0. After $\widehat{I_1'}$, there is no pending transaction that has a nontrivial conflict with T_0, and since the STM is 1-progressive, T_0 commits when executed in $\widehat{\alpha}'$ (Figure 2(b)); by indistinguishability, T_0 also commits when executed in α' (Figure 2(a)).

Consider now the execution $\alpha = I_1 T_0$, such that in I_1, p_1 executes a prefix of a transaction T_1 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(c)).

Since reads are invisible, α is indistinguishable to p_0 from an execution $\widehat{\alpha} = \widehat{I_1} T_0$, in which p_1 executes a prefix $\widehat{I_1}$ of a transaction $\widehat{T_1}$ with an empty read set writing to t_i the same value as T_1 until $\widehat{T_1}$ is logically committed (see Figure 2(d)), followed by T_0.

The item t_i is not in the data set of T_0 when executed in α or $\widehat{\alpha}$. This is clear when the data set of T_0 is static; the dynamic case is handled in [20].

Since the STM is oblivious, T_0 commits when executed after $\widehat{I_1}$ in $\widehat{\alpha}$ (Figure 2(d)), as it does after $\widehat{I_1'}$ in $\widehat{\alpha}'$ (Figure 2(b)). By indistinguishability, T_0 also commits when executed after I_1 (Figure 2(c)).

Let $m_1^0, \dots, m_k^0$ be the base objects that are private to p_0 after α. Since the STM is 1-progressive, T_1 commits when completed after α. 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 (Figure 2(e)).

Claim 3: p_1 modifies m_i^0 in J_1.

Proof: Since the STM is not eager, p_1 does not modify m_i^0 in I_1.

Process p_0 does not modify m_i^0 in α'. Otherwise, a nontransactional, uninstrumented read of t_i of p_0 after α' returns a value that is not the initial value of m_i^0, whereas no committed transaction writes to t_i in α', contradicting our weak safety property.

The executions $\widehat{\alpha}'$ and $\widehat{\alpha}$ are indistinguishable to p_0, since the STM is oblivious and since u or t_i are not in the data set of T_0. Therefore, the executions α' and α are indistinguishable to p_0 and hence, p_0 does not modify m_i^0 also in α.

It remains to show that p_1 modifies m_i^0 in J_1. Otherwise, a nontransactional read of t_i by p_0 after $I_1 T_0 J_1$ returns the initial value of m_i^0, since nontransactional reads are uninstrumented. This contradicts privatization safety, since T_1, which writes to t_i a value that is different from the initial value of m_i^0, commits before the nontransactional read of p_0. □

Therefore, p_1 applies a nontrivial primitive to m_i^0 in J_1 after the execution of T_0, in contradiction to privatization safety, which proves the lemma. □

We emphasize that in this theorem, the read sets of T_1 and T_1' are not empty, and thus, p_1 reads from the memory and may observe values written by p_0. However, because the STM has invisible reads, this looks to p_0 as if T_1 and T_1' have empty read sets, which suffices for the proof.

2. This can be done since STM is targeted towards generic, arbitrary concurrent applications and not only for custom-tailored concurrent data structures. In particular, STM should allow to compose transactions on more than a single data structure into an atomic computation; e.g., when removing a node from one linked list and reinserting it into another linked list. Therefore, it is reasonable to assume that data items are not all contained in a single data structure and that additional transactions and data items can always be introduced.

4 Privatization with Visible Reads

A corresponding lower bound can be shown for STMs with visible reads, assuming they ensure some degree of parallelism. The cost is stated in terms of low-level primitives by the privatizing transaction, rather than in terms of size of the privatizing transaction's data set. The result holds for any workload satisfying the condition specified in the theorem.

Theorem 4: For any privatization-safe STM that is l-progressive and oblivious, and every workload including a transaction T_0 privatizing items $t_1, \ldots t_k$ and k updating transactions $T_1, \ldots T_k$, such that transaction T_i writes to t_i and does not read from any item t_j, $j \neq i$ (it may read any other item), there is an execution in which T_0 accesses $r = \min\{l, k\}$ base objects.

While there are some similarities between the proof of this theorem and the proof of Theorem 2, the details are more involved, in order to accommodate visible reads. Therefore, before presenting the formal proof, we describe a high-level outline for a workload similar to the linked list of Figure 1, and then show how this counter-example plays out for TLRW [11].

4.1 Overview of the Proof and an Example for TLRW

Consider the scenario described on the left side of Figure 3: $R_0, R_1, \ldots, R_k$ are linked-list nodes (R_0 is the head node); each node R_i, $1 \leq i \leq k$, points to an item t_i.

We have k *updating* transactions traversing the nodes of a linked list ($R_0, R_1, \ldots, R_k$), while each transaction writes to a different item pointed by the list (i.e., the i-th transaction writes to t_i); each transaction writes to an item that is not read by other transactions, so these transactions do not have nontrivial conflicts. Later, a transaction by another process, privatizing all items pointed by the linked list ($t_1, \ldots t_k$), 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 transactions from the privatizing transaction. The challenge is to create an execution in which an updating transaction runs long enough to guarantee that it will 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 even starts.

It may seem that, at this point, the privatizing transaction does not need to have operations on many objects to observe a nontrivial conflict with the updating transactions, and it can abort or at least block until the situation is resolved. However, the obliviousness and non-eagerness of the STM can be used to "hide" the updating transactions from the privatizing transaction.

To employ obliviousness, we create a copy of the linked list, having exactly the same structure, which is not connected to the first list in any way.[3] It contains $k+1$ nodes $R'_0, R'_1, \ldots, R'_k$ and k items $t'_1, \ldots, t'_k$. The privatizing transaction does not have operations on this list at all; however, *shadow* transactions, without nontrivial conflicts, have operations on this completely disjoint linked list. Due to obliviousness, these transactions are indistinguishable from—and therefore can be swapped with—the original updating transactions.

We start with an execution in which the shadow transactions run one after the other. Then, we swap shadow transactions with updating transactions. Swapping is done inductively: Each iteration of our inductive construction swaps one shadow transaction with an updating transaction by the same process; that is, at each inductive iteration, one additional process executes an updating transaction instead of a shadow transaction, and *causes an access to at least one additional base object* by the privatizing transaction. This eventually yields an execution in which the privatizing transaction accesses many base objects, implying the lower bound.

A key technical challenge in the proof is in deciding which transaction to swap next, so as not to lose the events by the privatizing transaction that appear in the execution we have created so far. Specifically, we need to pick a shadow transaction T' such that swapping T' with the corresponding updating transaction T is invisible to the privatizing transaction in its execution prefix, at least during the events incurred due to previous iterations of the construction. This is done by choosing T to 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 read by the privatizing transaction in its execution prefix. (This is the purpose of Item 6 maintained by the inductive construction in the proof.)

Progressiveness is used to ensure that if at some point the privatizing transaction observes a nontrivial conflict, the updating transaction causing the conflict may run to completion. The progressiveness level should be high enough to enforce each updating transaction to complete, leading to a lower bound that is the minimum between the progressiveness and the number of privatized items. Progressiveness also ensures that the privatizing transaction runs to completion.

Consider this workload under TLRW, which is oblivious, as explained in Example 1 We consider a TLRW implementation with $n = k+1$ slotted threads, to be consistent with the prior discussion; in this example, one thread executes a privatizing transaction by setting the value of R_0 to null, and k threads execute transactions that write to a single item either in the left linked list or the right one.

In the initial execution E_0, all updating transactions write to items in the shadow copy of the linked list (right). Each transaction, at its turn, traverses the shadow

3. As in the proof of Theorem 2, this relies on the fact that the STM is general purpose, and additional transactions and data items can always be introduced.

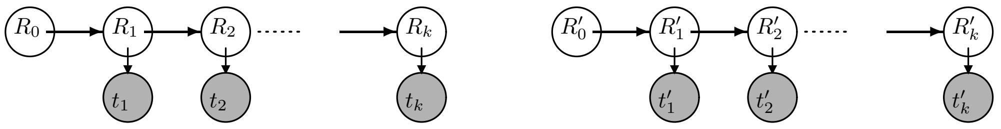


Fig. 3. An example workload for the proof of Theorem 4. The privatizing transaction privatizes the left linked list, while other transactions traverses either the left linked list or the right linked list, and write to respective items.

copy, reading the items in the path leading to the item on which the transaction acquires a write lock. More specifically, T_i' acquires the read lock on the root R'_0 by storing 1 in the i-th slot of this item, similarly T_i' acquires read locks on the items in the path to t'_i, and finally, T_i' sets the owner field of t'_i from $\perp$ to i, thus acquiring the write lock on it. At this point T_i' is logically committed, as it acquired all its locks, and since in TLRW transactions abort only due to time outs while attempting to acquire a lock. Note that T_i' has not completed yet, as it still needs to update the new value of t'_i. The suffix of E_0 is an execution of the privatizing transaction, writing null to R_0. T_0 sets the owner field of R_0 from $\perp$ to 0, then it spins until all the slots are 0. Since they are all zero, T_0 writes null to R_0 releases the lock and completes.

Next, we show what would happen if T_0 were not to read all k slots of the updating threads. For simplicity, assume that T_0 accesses slots in increasing order.

Let p_l be the thread with minimal identifier amongst the threads whose slots are not read by T_0 in E_0. We perturb E_0 to E_0^l so that, instead of executing T_l' in its turn, p_l executes the transaction T_l that traverses the left linked list and updates t_l, until p_l acquires the write lock on t_l and T_l is logically committed. This "swap" of T_l' with T_l is possible since the updating threads are oblivious to each other, and do not change their execution regardless of which linked list the other threads are updating. Note that in E_0^l T_l sets p_l's slot in R_0 to 1, so it is logically committed, but it has not completed yet, as it still needs to set a new value to t_l.

At this point, it should be clear why T_0 must read p_l's slot in R_0 in E_0^l and abort after spinning is timed out. Otherwise, T_0 commits also in E_0^l, privatizing all items in the linked list, including t_l. T_l, however, is still pending and will change t_l's value when it is permitted to resume its execution, thus violating privatization safety. To allow T_0 to commit (so we can continue with the proof) we further perturb E_0^l to E_1, such that during the execution of T_0, and immediately before T_0 accesses p_l's slot in R_0, T_l is permitted to be executed to completion, including setting p_l's slot in R_0 to 0.

We repeat this construction, selecting a thread from the threads whose slots are not read by T_0 in E_1. The assumption that T_0 accesses the slots in increasing order, and choosing the thread with minimal identifier, guarantee that the prefix of the execution of T_0 in E_1 (until it accesses p_l's slot in R_0) does not change during the following perturbations. In general, the prefix of T_0 that is constructed at a given iteration does not change in any of the following perturbations.

In this way, we show an execution in which T_0 must access all k slots of the updating transactions in R_0, otherwise privatization safety is violated.

4.2 Proof of Theorem 4

We now present the formal proof.

Consider $r+1$ processes $p_0, \ldots, p_r$, such that p_0 executes T_0, and p_i executes T_i for every i, $1 \leq i \leq r$. Consider shadow copies of all items in the data sets of $T_1, \ldots T_r$; $T_0, \ldots T_r$ do not have operations on any of these shadow items. Let $t'_1, \ldots t'_r$ be the shadow copies of $t_1, \ldots t_r$. For every process p_i, $1 \leq i \leq r$, consider an additional (shadow) transaction, T_i', reading the shadow copies of the items in the read set of T_i, and writing to the item t'_i, which is the shadow copy of t_i.

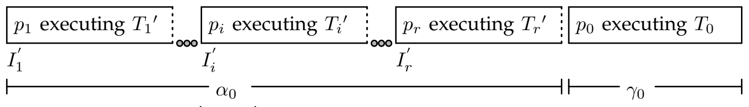
Note that for every i, T_i' does not read any item in $\{t_1, \ldots t_r\} \cup \{t'_1, \ldots t'_r\} \setminus \{t'_i\}$. Hence, the data sets of the transactions $T_1, \ldots, T_{j-1}, T_{j+1}, T_r, T_1', \ldots, T_r'$ do not have nontrivial conflicts. In addition, the data sets of the transactions $T_1, \ldots, T_r, T_1', \ldots, T_{j-1}', T_{j+1}', T_r'$ do not have nontrivial conflicts, for every j, $1 \leq j \leq r$. Also, T_i' does not read any item written by T_0; this ensures that T_i' and T_0 do not have nontrivial conflicts, for every i.

A process p *reads from* a process q *via* a base object o in an execution α if p accesses on 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.

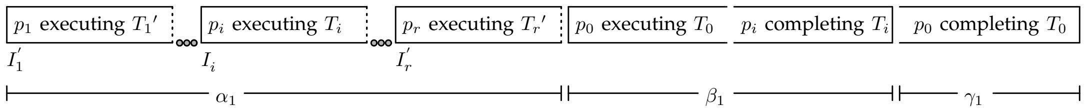
Consider the following execution $\alpha_0\beta_0\gamma_0$: α_0 is $I_1^{'} \ldots I_r^{'}$, such that in $I_i^{'}$, p_i executes a prefix of the transaction T_i' and T_i' is logically committed after $I_i^{'}$; β_0 is the empty execution interval; and γ_0 is a solo execution of T_0 by p_0 to completion (see Figure 4(a)).

For every ℓ, $0 < \ell \leq r$, 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) α_ℓ is a r-independent execution.
2) p_0 executes T_0 to successful completion in $\beta_\ell\gamma_\ell$.
3) p_0 reads from all processes in P_ℓ, a subset of $\{p_1, \ldots, p_r\}$ of size (at least) ℓ, in $\alpha_\ell\beta_\ell$.
4) 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.
5) Every process p_j, $\{p_1, \ldots, p_r\} \setminus Q_\ell$, executes $I_j^{'}$ in α_ℓ.
6) For every process p_j from which p_0 does not read in $\alpha_\ell\beta_\ell\gamma_\ell$, α_ℓ^j is a r-independent execution, in which p_j executes I_j instead of $I_j^{'}$, and all other processes



(a) The execution $\alpha_0\beta_0\gamma_0$: α_0 is $I_1' \ldots I_r'$; β_0 is the empty execution interval; and γ_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' in α_1; in β_1, p_0 starts executing T_0 and p_i completes T_i; p_0 completes T_0 in γ_1.

Fig. 4. The executions used in the proof of Theorem 4. A dotted line indicates that the transaction is logically committed.

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, $h < j$, and in $\alpha_\ell^j\beta_\ell$, p_0 does not read from p_j.

For $\ell = r$, we get an execution $\alpha_\ell\beta_\ell\gamma_\ell$, such that p_0 reads from r different processes in P_r (Condition 3), and hence, accesses r different base objects, implying the theorem.

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

1) α_0 is a r-independent execution, by construction.
2) Since all the transactions in α_0 do not have nontrivial conflicts with T_0, and since the STM is r-progressive ($r \leq l$), T_0 completes successfully in γ_0.
3) β_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.
4) Q_0 is empty, and the condition vacuously holds.
5) Every process $p_j \in \{p_1, \ldots, p_r\} \setminus Q_0 = \{p_1, \ldots, p_r\}$ executes T_j'; there are no nontrivial conflicts in this workload and since the STM is r-progressive, $\alpha_0 = I_1' \ldots I_r'$, such that p_i executes in I_i', a prefix of the transaction T_i' and T_i' is logically committed after I_i', is an execution interval.
6) For every process p_j from which p_0 does not read, α_0^j is a r-independent execution, as all the transactions do not have nontrivial conflicts, and since the STM is oblivious, α_0^j is an execution interval. Furthermore, β_0 is empty, thus, for every j, p_j does not modify in α_0^j any base object o that is accessed by p_0 in β_0, and in $\alpha_0^j\beta_0$ p_0 does not read from p_j since β_0 is empty.

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 $\{p_1, \ldots, p_r\} \setminus P_{\ell-1}$ 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, we pick a process from $V_{\ell-1}$ and swap its shadow transaction with its updating transaction, in order to construct the next step. For example, for $\ell = 1$, Figure 4(b) shows what happens if p_i is chosen from V_0, and its transaction is swapped to construct $\alpha_1\beta_1\gamma_1$.

Some care is needed when choosing the next process whose transaction is swapped, so as not to lose previous accesses by p_0 (accumulated in Condition 3); this is done by considering each process p_j still executing a shadow transaction, and choosing the first one that p_0 reads from.

In more detail, for any process $p_j \in V_{\ell-1}$, 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 r-independent, by construction and since the transactions do not have nontrivial conflicts. Since the STM is r-progressive, I_j (the prefix until T_j is logically committed) is a finite execution interval in $\alpha_{\ell-1}^j$. Since the STM is oblivious, $\alpha_{\ell-1}^j$ is an execution that is indistinguishable from $\alpha_{\ell-1}$ to every process in $\{p_1, \ldots, p_r\} \setminus \{p_j\}$. Furthermore, by the inductive assumption (Condition 6), 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, $h < j$, and p_0 does not read from p_j. Thus, p_0 reads the same values as in the execution of $\beta_{\ell-1}$, and the execution $\alpha_{\ell-1}^j\beta_{\ell-1}$ is indistinguishable to p_0 from $\alpha_{\ell-1}\beta_{\ell-1}$.

Consider the execution $\alpha_{\ell-1}^j\beta_{\ell-1}\gamma_{\ell-1}^j$, where p_0 runs solo in $\gamma_{\ell-1}^j$. Assume by way of contradiction that p_0 does not read from p_j also in $\alpha_{\ell-1}^j\beta_{\ell-1}\gamma_{\ell-1}^j$, then p_0 takes the same steps in $\gamma_{\ell-1}^j$ and $\gamma_{\ell-1}$ and T_0 is committed in $\alpha_{\ell-1}^j\beta_{\ell-1}\gamma_{\ell-1}^j$. Let $m_1, \ldots, m_r$ be base objects corresponding to the items $t_1, \ldots, t_r$ that are private to p_0 after $\alpha_{\ell-1}^j\beta_{\ell-1}\gamma_{\ell-1}^j$ (we omit the superscript 0).

The pending transactions in the execution $\alpha_{\ell-1}^j\beta_{\ell-1}\gamma_{\ell-1}^j$ do not have nontrivial conflicts. Since the STM is r-progressive and T_j is logically committed after I_j, T_j commits (writing to t_j) when executed solo after $\alpha_{\ell-1}^j\beta_{\ell-1}\gamma_{\ell-1}^j$. Consider the execution $\alpha_{\ell-1}^j\beta_{\ell-1}\gamma_{\ell-1}^j J_j$, such that J_j is the execution of T_j until it commits (see Figure 2(e)). In a manner similar to Claim 3, we show that p_j must modify m_j in J_j.

Claim 5: p_j modifies m_j in J_j.

Proof: Since the STM is not eager, p_j does not apply nontrivial primitive to m_j in $\alpha_{\ell-1}^j$ or in $\alpha_{\ell-1}$.

If p_0 modifies m_j in $\alpha_{\ell-1}\beta_{\ell-1}\gamma_{\ell-1}$, then a nontransactional read of 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 privatization safety.

Process p_0 does not read from p_j also in $\alpha^j_{\ell-1}\beta_{\ell-1}\gamma^j_{\ell-1}$, and it has the same steps in $\beta_{\ell-1}\gamma^j_{\ell-1}$ as in $\beta_{\ell-1}\gamma_{\ell-1}$. Thus, p_0 does not modify m_j also in $\alpha^j_{\ell-1}\beta_{\ell-1}\gamma^j_{\ell-1}$.

It remains to show that p_j modifies m_j in J_j. Otherwise, a nontransactional read of t_j by p_0 after $\alpha^j_{\ell-1}\beta_{\ell-1}\gamma^j_{\ell-1}J_j$ returns the initial value of m_j, since reads are uninstrumented. This contradicts privatization safety 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. $\square$

Therefore, p_j applies a nontrivial primitive to m_j in some step in J_j, after the execution of T_0, in contradiction to privatization safety. Thus, p_0 must read from p_j in $\alpha^j_{\ell-1}\beta_{\ell-1}\gamma^j_{\ell-1}$.

Let s_j be the number of steps until p_0 reads from p_j for the first time in $\gamma^j_{\ell-1}$. Let $j_\ell = j$ such that s_j is the smallest, i.e., p_0 reads from p_j the earliest; if $s_j = s_h$, i.e., p_0 reads from p_j and p_h after the same number of steps, then assume that $j > h$, i.e., p_j appears after p_h in the execution interval α_ℓ.

Let the execution interval α_ℓ be $\alpha^{j_\ell}_{\ell-1}$. The execution interval β_ℓ is $\beta_{\ell-1}$ extended with the first $s_{j_\ell}-1$ steps of p_0 in $\gamma^{j_\ell}_{\ell-1}$, then a solo execution of p_{j_ℓ} completing T_{j_ℓ}, and finally, the s_{j_ℓ}-th step of p_0 from $\gamma^{j_\ell}_{\ell-1}$, which reads from p_{j_ℓ}. Since T_{j_ℓ} is logically committed in α_ℓ, and the STM is r-progressive, T_{j_ℓ} commits in β_ℓ. The execution interval γ_ℓ is the solo execution of p_0 completing T_0.

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

1) α_ℓ is a r-independent execution, by construction.
2) T_0 completes successfully since there is no pending transaction with a nontrivial conflict after $\alpha_\ell\beta_\ell$, and the STM is r-progressive.
3) By the inductive assumption (Condition 3), 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 every processes p_h, if $h < j_\ell$. Furthermore, since the STM is oblivious, $\alpha_{\ell-1}$ and α_ℓ are indistinguishable to every process p_h, if $h > j_\ell$. 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, $P_\ell \supseteq P_{\ell-1} \mathbin{\dot\cup} \{p_{j_\ell}\}$, and $|P_\ell| \geq |P_{\ell-1}| + 1 \geq \ell$.
4) By the inductive assumption (Condition 4), $Q_{\ell-1}$ 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 processes in $\{p_1,\ldots,p_r\} \setminus \{p_{j_\ell}\}$, and only p_{j_ℓ} switched from I'_{j_ℓ} in $\alpha_{\ell-1}$ to I_{j_ℓ} in α_ℓ, every process $p_h \in Q_\ell \setminus \{p_{j_\ell}\}$ executes I_h in α_ℓ and completes T_h in $\beta_{\ell-1}$, which is a prefix of β_ℓ.
5) By the induction assumption (Condition 5), every process $p_h \in \{p_1,\ldots,p_r\}\setminus Q_{\ell-1}$ executes I'_h in $\alpha_{\ell-1}$. Hence, every process $p_h \in \{p_1,\ldots,p_r\}\setminus Q_\ell$ executes I'_h in α_ℓ, since only $p_{j_\ell} \notin \{p_1,\ldots,p_r\} \setminus Q_\ell$ switched from I'_{j_ℓ} in $\alpha_{\ell-1}$ to I_{j_ℓ} in α_ℓ.
6) Assume, by way of contradiction, that for some j, p_j modifies in α^j_ℓ the base object o, which in $\alpha_\ell\beta_\ell$ p_0 reads from p_h, $h < j$, in step σ. Consider iteration ℓ_h in which the transaction executed by p_h is switched from the shadow transaction $T_h{}'$ to the updating transaction T_h. Since the STM is oblivious, the executions α_{ℓ_h} and α_ℓ are indistinguishable to p_j. Therefore, p_j modifies the base object o also in $\alpha^j_{\ell_h}$, and in $\alpha^j_{\ell_h}\beta_{\ell_h}$, p_0 reads from p_j no later than step σ. But this is a contradiction, since the rule for choosing the transaction to switch dictates that p_j should have been chosen instead of p_h, because in iteration ℓ_h, $s_j \leq s_h$ and $j > h$. This also implies that in $\alpha^j_\ell\beta_\ell$, p_0 reads the same values in the same base objects from the same processes as in $\alpha_\ell\beta_\ell$, and in particular, p_0 does not read from p_j in $\alpha^j_\ell\beta_\ell$.

4.3 Reducing the Cost of Privatization by Limiting Parallelism

The previous lower bound, stated as the minimum between the number of privatized items and the level of parallelism, points out a way to reduce the cost of privatization, namely, to limit the parallelism offered by the STM. We next show how this tradeoff is indeed tight, by sketching a "counter-example" STM, a variant of RingSTM [34] called *VisibleRingSTM*, which reduces the cost of privatization by limiting parallelism.

RingSTM is oblivious and privatization safe, but not progressive; privatizing k items requires $O(c)$ steps, 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 used only locally by the transaction. On 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 nontrivial 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. In RingSTM, a transaction blocks until all concurrent logically committed transactions are completed, therefore RingSTM is not l-progressive, for any $l \geq 1$.

In VisibleRingSTM, we make the read set Bloom filter also visible to other transactions, just like the write set filter. 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, it checks for intersections with the write sets of these transactions, like RingSTM. 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.

The steps of the commit operation of a transaction T in VisibleRingSTM are:

1) Check intersection of the read set filter with the write set filters of concurrent transactions preceding T in the ring (validating the read set),
2) If there is a read-after-write conflict, then abort.
3) Commit T in the ring; if not successful start again from 1 (otherwise, T is logically committed).
4) Check intersection of the write set filter with the write set *and read set* filters of concurrent transactions preceding T in the ring,
5) Wait until all preceding transactions with write-after-write and *write-after-read* conflicts are completed, and then complete T.

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 reads the c ring entries of concurrent logically committed transactions, the items in its data set and the global ring index.

Using a ring of size c_0 can bound the cost of a privatizing transaction by $O(c_0)$, for any $c_0 > 1$, since it has to read 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 finds an empty entry, becomes logically committed, and continues as in VisibleRingSTM. This variant 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, privatization cost is reduced by limiting the progress of concurrent transactions.

5 Disjoint-Access Parallel STMs

Disjoint-access parallelism [21] is a property that captures the intuition that transactions operating on disjoint parts of the data should not interfere with each other [19]. This section discusses the relationship between oblivious and disjoint-access parallel STMs.

To define disjoint-access parallelism somewhat more precisely (cf. [4]), we represent (trivial and nontrivial) conflicts between overlapping transactions in some execution interval, with a *conflict graph*: The vertices represent transactions and there is an edge between two overlapping transactions that have operations on the same item; these could be two read operations. 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. We say that two transactions *contend* if they have pending primitives, at least one of which is nontrivial, on the same base object at some configuration. An STM is *(weakly) disjoint-access parallel* if two processes executing transactions T_1 and T_2 contend only if T_1 and T_2 are not disjoint access.

We argue that with invisible reads, disjoint-access parallel STMs are oblivious. Disjoint-access parallelism ensures that transactions that are not connected in the conflict graph do not concurrently contend on a (or even access the same) base object. Note that obliviousness enforces a certain behavior of a transaction T only after a k-independent execution without nontrivial conflicts with T. To show that disjoint-access parallel STMs with invisible reads are oblivious, we only need to consider these particular executions.

Consider a transaction T by process p_0 and two k-independent executions, α_1 and α_2, in which none of the transactions has a nontrivial conflict with T. Consider a perturbation of the execution α_1 in which all write operations write the same values, however all transactions have empty read sets; call this execution α'_1. Reads invisibility implies that the execution $\alpha_1 T$ is indistinguishable to p_0 from the execution $\alpha'_1 T$. The same is true for $\alpha_2 T$ and $\alpha'_2 T$, in which all transactions have empty read sets. Next, consider the conflict graphs of T in the executions $\alpha'_1 T$ and $\alpha'_2 T$. In both executions, the data set of T does not intersect with any of the data sets of the transactions preceding it, therefore T is isolated in both conflict graphs. Disjoint-access parallelism implies that in both executions T does not access any base object to which the transactions preceding it has accessed, and therefore the execution interval of T in both executions is the same. This means that $\alpha'_1 T$ and $\alpha'_2 T$ are indistinguishable to p_0, and by the invisibility of reads, $\alpha_1 T$ and $\alpha_2 T$ are indistinguishable to p_0.

Going back to the definition of obliviousness, and since we considered an arbitrary pair of k-independent executions, this implies the implementation is oblivious. Hence, Theorem 2 implies:

Corollary 6: For any privatization-safe STM that is 1-progressive, disjoint-access parallel and with invisible reads, there are privatization workloads, for which there is an execution where the size of the data set of a transaction privatizing k items is $\Omega(k)$.

The specific workloads that incur the cost stated in Corollary 6 are defined in the proof of Theorem 2. In these workloads transactions have nonempty read sets.

Some clock-based STMs [10], [28], [29] are not disjoint-access parallel, but they are oblivious and have invisible reads, so our lower bound holds for them. This shows the importance of proving the lower bound under the weaker condition of obliviousness, rather than disjoint-access parallelism as in most prior lower bounds.

When reads are visible, $\alpha_1 T$ and $\alpha'_1 T$ cannot be swapped, and we need a notion of disjoint-access parallelism that takes into account only nontrivial conflicts. An edge in a *nontrivial conflict graph* connects two *overlapping* 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.

To show that nontrivial disjoint-access parallel STMs are oblivious we consider again an arbitrary pair of k-independent executions α_1 and α_2, followed by a nonconflicting transaction T. Instead of swapping α_1 and α_2 with executions with empty read sets, we note that T is isolated in the nontrivial conflict graphs of the executions $\alpha_1 T$ and $\alpha_2 T$. By nontrivial disjoint-access parallelism, the execution interval of T in both executions is the same, and the implementation is oblivious. By Theorem 4:

Corollary 7: For any privatization-safe STM that is l-progressive and nontrivial disjoint-access parallel there are privatization workloads (defined in Theorem 4) in which update transactions have nonempty read sets, for which there is an execution where a transaction privatizing k items accesses $\min\{l, k\}$ base objects.

6 Related Work

Lower bounds, like those proved in this paper, usually rely on certain assumptions, and hence, a common way to circumvent the lower bounds is not to satisfy these assumptions. This section discusses recent STMs supporting privatization, relating them to our lower bounds and the assumptions they rely on—obliviousness, visibility of reads and progressiveness; this is summarized in Table 1. Later in this section, we discuss other approaches to combining transactional and nontransactional accesses.

As we show in detail below, many STMs supporting privatization are oblivious, due to the cost of tracking the read sets of other, non-conflicting transactions. As we can see from Table 1, most of these STMs avoid the lower bounds by limiting parallelism, being 0-progressiveness.

An interesting exception is TLRW [11], which is oblivious when restricted to slotted readers (see Example 1). In accordance with our lower bound, the number of locations read by a privatizing transaction is linear in the number of slotted readers. Although these locations are packed together, they may generate a lot of cache traffic when updated by many threads, depending on the cache consistency protocol and the workload.

Some oblivious STMs have invisible reads [8], [26], [34], making their read set transparent for other transactions; others, e.g., [9], have *partially visible* reads [24], meaning that other transactions cannot determine which transaction is reading the item; there are also oblivious STMs with visible reads, e.g., [14], however, their execution is unaffected by trivial, read-read conflicts.

Many oblivious STMs with visible reads, which support privatization, e.g., [8], [14], [24], [26], [27], [34], are 0-progressive. Studying these STMs, as we do next, reveals that despite different implementations mechanisms, they all sacrifice parallelism by using global synchronization mechanisms. These mechanisms force the order by which transactions commit, while preventing other transactions to make progress and commit.

JudoSTM [27] transforms code to support transactional execution on-the-fly at run-time. The coarse-grained commit variant uses single lock; when a transaction commits its changes, no concurrent transaction can commit and complete. NOrec [8], like JudoSTM, uses a single sequence lock yet it is livelock-free—a transaction aborts only due to a nontrivial conflict with concurrent, logically committed transaction.

As described in Section 4.3, RingSTM [34] uses Bloom filters to represent the read and write sets of transactions. Appending an entry to a global ring effects a logical commit of a writing transaction. A logically committed transaction may block due to nonconflicting concurrent logically committed (write) transactions. The default variant of RingSTM is livelock-free, permitting concurrent disjoint transactions to commit their changes in parallel. The single-writer variant forbids committing the changes of concurrent transactions in parallel, even of disjoint transactions.

Menon et al. [26] suggest two STMs that provide privatization safety. The first uses a global linearization timestamp. It enforces start and commit linearization, meaning the start, commit, and completion order of transactions are the same. Alternatively, ordering is imposed only among conflicting transactions, however, the completion order is similar to the commit order in all transactions. In both cases, this is done by iterating over other processes with concurrent transactions, waiting for their completion.

Marathe et al. [24] use a globally synchronized clock and a linked list containing all active transactions to guarantee transaction consistency. If a transaction identifies a nontrivial conflict with a concurrent transaction it blocks until all active transactions complete.

InvalSTM [14] invalidates at commit time: a transaction invalidates all concurrent transactions with which it has nontrivial conflicts. Like RingSTM, Bloom filters are used to compress the read and write sets of a transaction. To commit, a transaction acquires global locks preventing o ther transactions from committing or making any progress or new transactions to begin.

Other STMs [11], [23] are progressive, but they are not oblivious. SkySTM [23] is designed to work in hybrid transactional memory systems, combining software and hardware. It is not oblivious, as it uses a global counter to indicate when a writing transaction, which has a nontrivial conflict with a reader, commits. It also has partially visible reads. A transaction blocks only due to conflicting transactions and aborts only due to a nontrivial, read-write conflict with another transaction. TLRW [11], when considering non-slotted readers, is a lock-based STM, in which a transaction aborts only due to a conflict with another transaction and blocks only while waiting for a lock on an item.

All the STMs discussed so far, support *implicit privatization* [24], meaning that the implementation is re-

	Oblivious	**Visible reads**	**Progressiveness**	**Cost**
JudoSTM: Olszewski et al. [27]	yes	no	0	$O(1)$
RingSTM: Spear et al. [34]	yes	no	0	$O(1)$
Menon et al. [26]	yes	no	0	$O(1)$
NOrec: Dalessandro et al. [8]	yes	no	0	$O(1)$
Marathe et al. [24]	yes	partially	0	$O(\#\text{ processes})$
Private transactions: Dice et al. [9]	yes	partially	0	$O(1)$
InvalSTM: Gottschlich et al. [14]	yes	yes	0	$O(1)$
VisibleRingSTM (this paper)	yes	yes	c_0	$O(c_0)$
TLRW, slotted readers: Dice and Shavit [11]	yes	yes	# slotted readers	$O(\#\text{ slotted readers})$
TLRW: Dice and Shavit [11]	no	partially	# slotted readers	$O(\#\text{ slotted readers})$
SkySTM: Lev et al. [23]	no	partially	# processes	$O(1)$

TABLE 1
Comparison of privatization-safe STMs, showing obliviousness, visibility of reads, progressiveness, and cost—measured in number of memory locations accessed.

quired to handle all transactions as if they are potentially privatizing items; this incurs excessive overhead for all transactions. In *explicit privatization* [32], 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 [23].

Some experiments [12], [32], [37] tested techniques for 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 even worse than in sequential code.

Guerraoui et al. [15] define *parameterized opacity*, a framework, extending *opacity* [16], for modeling the interaction between transactions and nontransactional operations. The model is parameterized with a memory model, capturing 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. They prove that parameterized opacity cannot be achieved if the memory model restricts the order of read or write operations to different variables. Furthermore, parameterized opacity requires either instrumenting nontransactional operations or using RMW primitives when writing inside a transaction, if the memory model allows reordering operations to different variables. 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. It can be shown [20] that an oblivious uninstrumented STM, that is 1-progressive, i.e., allows more than one transaction to proceed concurrently, does not provide opacity parameterized with respect to any memory model. These results assume that items are accessed nontransactionally without a preceding privatization transaction, and show the implications of *not privatizing*, while our results complete the picture by showing the *cost of privatization*.

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 property follows from parameterized opacity [16], regardless of the memory model imposed on nontransactional reads and writes.

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 until all active transactions are completed; thus other, non-privatizing transactions avoid the overhead of privatization. The barrier reads an array whose size is proportional to the maximal parallelism, demonstrating again the inherent tradeoff we have proved between parallelism and privatization cost, in oblivious STMs. As in VisibleRingSTM, the size of the active transactions array can be bounded, thereby reducing the overhead of the barrier, but at the cost of limiting the level of parallelism.

An alternative way to provide strong atomicity is thorough *static separation* [2]. This is a discipline in which transactional and nontransactional accesses are not mixed on the same data item. To access items nontransactionally, a transaction copies them to a private buffer, trivially incurring the cost predicted by our lower bound. *Dynamic separation* [1] allows data to change modes without being copied, simply by setting a protection mode in the item. Dynamic separation requires to modify the protection mode of all items to become privatized, in accordance with our lower bound.

7 Discussion

This paper studies the theoretical complexity of privatization with uninstrumented nontransactional reads, and shows an inherent cost, linear in the number of privatized items. In STMs with invisible reads, a transaction privatizing k items must have a data set of size

k. A more involved proof shows that even when the STM has visible reads, the privatizing transaction must access r memory locations, where r is now the minimum between k, 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 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 [22], in which transactional writes interfere with nontransactional accesses, and the *doomed transaction* problem [35], in which transactional reads of private data lead to inconsistent state. Our definition of privatization safety (Property 4) formalizes the first problem; our results show that this problem by itself is an impediment to the desire to provide efficient privatization.

As discussed in Section 6, some STMs maintain visible reads, yet they are oblivious [9], [14]. SkySTM [23] has visible reads, and 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, because 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 takes $O(1)$ steps, and any writing transaction writes to a number of base objects that is linear in the size of its write set.

Strong privatization safety [23] 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, Sandeep Hans, Petr Kuznetsov, Alessia Milani, Michael Spear, Eran Yahav and the anonymous referees 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 *Proceedings of the 18th International Conference on Compiler Construction (CC)*, pages 63–77, 2009.

[2] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. *ACM Trans. Program. Lang. Syst.*, 33:2:1–2:50, January 2011.

[3] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 185–196, 2009.

[4] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. *Theory of Computer Systems*, 49(4):698–719, Nov. 2011.

[5] H. Avni and N. Shavit. Maintaining consistent transactional states without a global clock. In *Proceedings of the 15th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 131–140, 2008.

[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. *ACM SIGPLAN Notices*, 45(5):67–78, 2010.

[9] D. Dice, A. Matveev, and N. Shavit. Implicit privatization using private transactions. In *5th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2010.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, pages 194–208, 2006.

[11] D. Dice and N. Shavit. TLRW: return of the read-write lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 284–293, 2010.

[12] A. Dragojevic, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a research toy. *Commun. ACM*, 54(4):70–77, 2011.

[13] J. Duffy. A (brief) retrospective on transactional memory. http://www.bluebytesoftware.com/blog/, January 2010.

[14] J. E. Gottschlich, M. Vachharajani, and J. G. Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the 8th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 101–110, 2010.

[15] R. Guerraoui, T. A. Henzinger, M. Kapalka, and V. Singh. Transactions in the jungle. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 263–272, 2010.

[16] R. Guerraoui and M. Kapałka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 175–184, 2008.

[17] R. Guerraoui and M. Kapałka. The semantics of progress in lock-based transactional memory. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 404–415, 2009.

[18] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, pages 522–529, 2003.

[19] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[20] E. Hillel. *Concurrent Data Structures: Methodologies and Inherent Limitations*. PhD thesis, Technion, 2011.

[21] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th annual ACM symposium on Principles of Distributed Computing (PODC)*, pages 151–160, 1994.

[22] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[23] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2009.

[24] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *Proceedings of the 2008 37th International Conference on Parallel Processing (ICPP)*, pages 67–74, 2008.

[25] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17, 2006.

[26] 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 *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 314–325, 2008.

[27] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In

Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT), pages 365–375, 2007.
[28] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, pages 284–298, 2006.
[29] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the 19th annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 221–228, 2007.
[30] 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.
[31] 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.
[32] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Ordering-based semantics for software transactional memory. In *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS)*, pages 275–294, 2008.
[33] 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.
[34] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 275–284, 2008.
[35] 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 *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 34–48, 2007.
[36] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.
[37] 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 *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 265–274, 2008.

Hagit Attiya received the B.Sc. degree in Mathematics and Computer Science from the Hebrew University of Jerusalem, in1981, the M.Sc. and Ph.D. degrees in Computer Science from the Hebrew University of Jerusalem, in 1983 and 1987, respectively.
She is presently a professor at the department of Computer Science at the Technion, Israel Institute of Technology, where she holds the Harry W. Labov and Charlotte Ullman Labov Academic Chair. Before joining the Technion, she was a post-doctoral research associate at the Laboratory for Computer Science at M.I.T.
Her general research interests are in the area of distributed and parallel computation. She is the editor-in-chief of the journal Distributed Computing, and a fellow of the ACM.

Eshcar Hillel received the B.Sc. degree in Software Engineering (summa cum laude) and the Ph.D. degree in Computer Science from the Technion, Israel Institute of Technology, in 2002 and 2011, respectively.
In February 2011, she joined Yahoo! Labs, where she is involved in projects on large-scale distributed data storage systems such as Hadoop and HBase, as well as projects studying reinforcement learning in a distributed setting.
Prior to her graduate studies she worked at Hewlett-Packard (formerly Compaq labs), where she was involved in Non-Stop relational database system projects.