Snapshot Isolation Does Not Scale Either*

Victor Bushkov
EPFL, IC, LPD
victor.bushkov@epfl.ch

Dmytro Dziuma
FORTH-ICS
ddziuma@ics.forth.gr

Panagiota Fatourou
University of Crete & FORTH-ICS
faturu@csd.uoc.gr

Rachid Guerraoui
EPFL, IC, LPD
rachid.guerraoui@epfl.ch

1 Introduction

Transactional memory (TM) [11, 15] allows concurrent processes to execute operations on *data items* within atomic blocks of instructions, called *transactions*. A transaction may either *commit* in which case all its updates become visible to other transactions, or *abort* in which case all its updates are discarded. The paradigm is appealing for its simplicity but implementing it efficiently is challenging. Ideally, the TM system should not introduce any contention between transactions beyond that inherently due to the actual code of the transactions. In other words, if two transactions access disjoint sets of data items (i.e. if they do not *conflict*), then none of these transactions should delay the other one, i.e., these transactions should not *contend* on any base object. This requirement has been called *strict disjoint-access-parallelism*. *Base objects* are low-level objects, which provide atomic *primitives* like *read/write*, *load linked/store conditional*, *compare-and-swap*, used to implement the TM system. Two transactions *contend* on some base object if both access that object during their executions and one of them performs a *non-trivial* operation on that object, i.e. an operation which may update its state.

Disjoint-access-parallelism was introduced in [12]. Later variants [1, 2, 6] employed the concept of a conflict graph. A *conflict graph* is a graph whose vertices represent transactions (or operations) performed in an execution α and an edge exists between two nodes if the corresponding transactions (operations) conflict in α. In most of these definitions, *disjoint-access-parallelism* requires any two transactions to contend on a base object only if there is a path in the conflict graph of the minimal execution interval that contains both transactions. Different variants are met in the literature with the names disjoint-access-parallelism or weak disjoint-access-parallelism (most of them use different properties to restrict access to base objects). Stronger versions of disjoint-access-parallelism like strict disjoint-access-parallelism, result in more parallelism (and promote scalability) and therefore they are highly desirable when designing TM implementations: strict disjoint-access-parallelism is indeed ensured by *blocking* TM algorithms like TL [4]. Nevertheless, a transaction that locks a data item and gets paged out might block all other transactions for a long amount of time. One might require a liveness property that prevents such blocking. It was shown however in [8] that a TM cannot ensure strict disjoint-access-parallelism if it also needs to ensure *serializability* [13] and obstruction-freedom [7, 5]. *Obstruction-freedom* ensures that a transaction can be aborted only when step contention is encountered during the course of its execution. Obstruction-freedom is weaker than *lock-freedom* or *wait-freedom*. It allows for designing simpler TM algorithms and therefore it has been given special attention in TM computing [9].

We study the following question: can we ensure strict disjoint-access-parallelism and obstruction

*This work has been supported by the European Commission under the 7th Framework Program through the TransForm (FP7-MC-ITN-238639) project.

freedom if we consider other consistency conditions? In other words, is consistency a major factor against scalability? We focus on *snapshot isolation* [3], a safety property which, roughly speaking, requires that transactions should be executed as if every read operation reads from some snapshot of the memory that was taken when the transaction started. Snapshot isolation is an appealing property for TM computing since it provides the potential to increase throughput for workloads with long transactions [14]. If the set of transactions is restricted to those that do not read data items they have previously written, snapshot isolation is a weaker property than strict serializability.

We prove that the answer is still negative. Namely, it is impossible to implement a TM which is strict disjoint-access-parallel and satisfies obstruction-freedom and snapshot isolation. To make our impossibility result stronger, we consider, for its proof, a weak snapshot isolation property which requires only that each transaction reads from some consistent snapshot of the memory taken when it starts, thus ignoring the extra constraint (met in the literature [3, 14] for snapshot isolation) that from two concurrent transactions writing to the same data item, only one must commit. It is remarkable that our impossibility result holds for two variants of snapshot isolation, the original one from the database world where a read of a data item following a write to this data item by the same transaction returns the value written by this write, and a simpler one where all reads (even those following writes to the same data item by the same transaction) return the value recorded in the snapshot taken at their start.

We also show how to circumvent the impossibility result if we relax the disjoint-access-parallelism requirement between update transactions: mainly we show how we can get a simplified version of DSTM [5], called SI-DSTM, which satisfies snapshot isolation, obstruction-freedom, and the following weaker disjoint-access-parallelism requirement: two operations (executed by two concurrent transactions T_1 and T_2) on different data items, one of which is a read operation, never contend on the same base object, and two write operations (by T_1 and T_2) on different data items contend on the same base object only if there is a chain of transactions starting with the transaction that performs one of these write operations and ending with the transaction that performs the other, such that every two consecutive transactions in the chain *conflict* and the execution interval of each of these transactions overlaps with that of either T_1 or T_2. We call the property satisfied by write transactions *weak disjoint-access-parallelism*. DSTM satisfies weak disjoint-access-parallelism for all transactions, while SI-DSTM satisfies strict disjoint-access-parallelism for read-only transactions and weak disjoint-access-parallelism for write transactions. Moreover, SI-DSTM is significantly simpler than the original DSTM algorithm.

2 Useful Definitions

A TM algorithm provides implementations for the routines `readDI` and `writeDI` which are called to read or write data items, respectively, and for the routines `CommitTr`, and `AbortTr`, which are called when a transaction tries to commit or abort, respectively. Each time a *transaction* calls one of these routines we say that it *invokes* an *operation*; when the execution of the routine completes, a *response* is returned. We denote the invocation of `CommitTr` by a transaction T as $commit_T$; the response to $commit_T$ can be either C_T (commit) or A_T (abort). We denote by (i) $x.write(v)$ the invocation of `writeDI` for data item x with value v; it returns *ok* if the write was successful or A_T if the transaction that invoked it has to abort, (ii) $x.read()$ the invocation of `readDI` for data item x; it returns a value for x if the operation was successful or A_T if the transaction that invoked it has to abort.

A *configuration* consists of the state of each process and the state of each base object. A *step* of a process consists of applying a single operation on some base object, its response, and zero or more local operations that may cause the internal state of the process to change; each step is executed atomically. An *execution* α is a sequence of steps. An execution is *solo* if every step is performed by the same process. Two executions α_1 and α_2 starting from configurations C_1 and C_2, respectively, are *indistinguishable* to transaction T, if the state of T is the same in C_1 and C_2, and the sequence of

steps performed by T (and thus also the responses it receives) are the same during both executions.

A *history* H is a sequence of invocations and responses performed by transactions. Given an execution α, we denote by H_α the sequence of invocations and responses performed by the transactions in α. Let $H|T$ be the longest subsequence of H consisting only of invocations and responses of a transaction T. We say that T *commits* (*aborts*) in H if $H|T$ ends with C_T (A_T). If $H|T$ ends with a *commit* invocation, then T is *commit-pending*. A history H is *sequential* if no two transactions are concurrent in H. H is *complete* if it does not contain any transactions that have neither committed not aborted. A read operation $x.read()$ by some transaction T is *global* if T has not invoked $x.write(*)$ before invoking $x.read()$. Transaction T is *legal* in a *sequential* history H, if every read operation r whose response is not A_T, returns a value v such that: (i) if r is not global, then v is the value written by the last invocation of $x.write$ preceding r in H; (ii) if r is global and there are committed transactions preceding T in H which invoke $x.write(*)$, then v is the argument of the last $x.write()$ invocation by those transactions; (iii) otherwise, v is the initial value of x. A complete sequential history H is *legal* if every transaction is legal in H.

Let T be a committed or commit-pending transaction in a history H. Let $T|read$ be the longest subsequence of $H|T$ consisting only of the global read invocations and their corresponding responses and $T|other$ be the subsequence $H|T - T|read$, i.e. $T|other$ consists of all invocations performed by T (and their responses) other than those comprising $T|read$. Let λ be the empty execution. Then we define T_r and T_o in the following way:

- $T_r = T|read \cdot commit_{T_r} \cdot C_{T_r}$ if $T|read \neq \lambda$, and $T_r = \lambda$ otherwise, and
- $T_o = T|other \cdot commit_{T_o} \cdot C_{T_o}$ if $T|other \neq \lambda$, and $T_o = \lambda$ otherwise.

Definition 2.1. *An execution α satisfies* snapshot isolation*, if for* every committed *transaction T (and for* some *of the commit-pending transactions) in α it is possible to insert a read serialization point $*_{T,r}$ and a write serialization point $*_{T,w}$ such that: (i) $*_{T,r}$ precedes $*_{T,w}$, (ii) both $*_{T,r}$ and $*_{T,w}$ are inserted within the execution interval of T, and (iii) if σ_α is the sequence defined by these serialization points, in order, and H_{σ_α} is the history we get by replacing each $*_{T,r}$ with T_r and each $*_{T,w}$ with T_o in σ_α, then H_{σ_α} is legal.*

3 The impossibility result

Theorem 3.1. *No obstruction-free STM can ensure both snapshot isolation and strict disjoint-access-parallelism.*

Proof. We provide a brief description of the proof. Assume, by contradiction, that there is an obstruction-free STM which ensures both snapshot isolation and strict disjoint-access-parallelism. We consider the following transactions, all executed by distinct processes: (1) T_1, executed by process p_1, writes value 1 to a, b, and c, (2) T_2, executed by process p_2, writes value 2 to a, d, and e, (3) T_3, executed by process p_3, reads from b, (4) T_4, executed by process p_4, reads from d, (5) T_5, executed by process p_5, reads from e, (6) T_5', executed by process p_5', reads from c, and (7) T_6, executed by process p_6, reads from a.

We will construct two executions: $\delta = \alpha_1 \cdot \alpha_2 \cdot s_1 \cdot s_2 \cdot \gamma_6$, and $\delta' = \alpha_1 \cdot \alpha_2 \cdot s_2 \cdot s_1 \cdot \gamma_6'$, such that: (i) γ_6 and γ_6' are solo executions of T_6, (ii) in γ_6, T_6 reads the value 2 for a, (iii) in γ_6', T_6 reads the value 1 for a, and (iv) γ_6 and γ_6' are indistinguishable. Thus, T_6 should read the same value for a in both executions which is a contradiction.

Specifically, α_1 is the solo execution of T_1 by process p_1 from the initial configuration C_0 up until the point that p_1 is poised to execute step s_1 which is *critical* for T_3, i.e. in the execution γ_3' where p_3 executes solo T_3 before s_1 until T_3 commits (T_3 will indeed commit because of snapshot isolation) T_3 reads 0 for b, whereas in the execution γ_3 where p_3 executes solo T_3 after s_1 until T_3 commits, T_3 reads 1 for b. To define α_2 and s_2, we let p_2 execute solo T_2, after α_1, until it is poised to execute

a step which is critical for T_4, i.e. in the execution γ'_4 where p_4 executes solo T_4 before s_2 until T_4 commits, T_4 reads 0 for d, whereas in the execution γ_4 where p_4 executes solo T_4 after s_2 until T_4 commits, T_4 reads 2 for d.

We argue that (i) s_1 applies a non-trivial operation on some base object o_1 such that T_3 reads o_1 in γ_3, (ii) γ_3 is legal after $\alpha_1 \cdot \alpha_2 \cdot s_1$ and γ'_3 is legal after $\alpha_1 \cdot \alpha_2 \cdot s_2$, and (iii) γ_3 and γ'_3 do not read any base object written by $\alpha_2 \cdot s_2$. Similarly, we argue that (i) s_2 applies a non-trivial operation on some base object $o_2 \neq o_1$ such that T_4 reads o_2 in γ_4, (ii) γ_4 and γ'_4 are legal after $\alpha_1 \cdot \alpha_2 \cdot s_1$, and (iii) γ_4 and γ'_4 do not read any base object written by $\alpha_1 \cdot s_1$.

To argue that γ_6 and γ'_6 will return the required values, we consider the following two executions: $\gamma = \alpha_1 \cdot \alpha_2 \cdot s_1 \cdot \gamma_3 \cdot \gamma'_4 \cdot s_2 \cdot \gamma_5 \cdot \gamma_6$ (Fig. 1) and $\gamma' = \alpha_1 \cdot \alpha_2 \cdot s_2 \cdot \gamma_4 \cdot \gamma'_3 \cdot s_1 \cdot \gamma'_5 \cdot \gamma'_6$, where γ_5 is the solo execution of T_5 by p_5 until T_5 commits, and γ'_5 is the solo execution of T'_5 by p'_5 until T'_5 commits. We argue that these executions are legal. To conclude the proof, we prove that γ and δ are indistinguishable to p_6 and T_6 reads 2 for a in γ; similarly, γ' and δ' are indistinguishable to p_6 and T_6 reads 1 for a in γ'. □

We remark that the imposibility result also holds for a simpler version of Definition 2.1 where $T|read$ contains all read operations (global or not) performed by T and $T|other$ contains all its write operations. moreover, the impossibility result holds if we consider a stronger version of Definition 2.1 which ensures the stated property for every prefix α' of *alpha*. Finally, we have shown that the impossibility result holds even if the system provides primitives that *atomically* access more than one base objects.

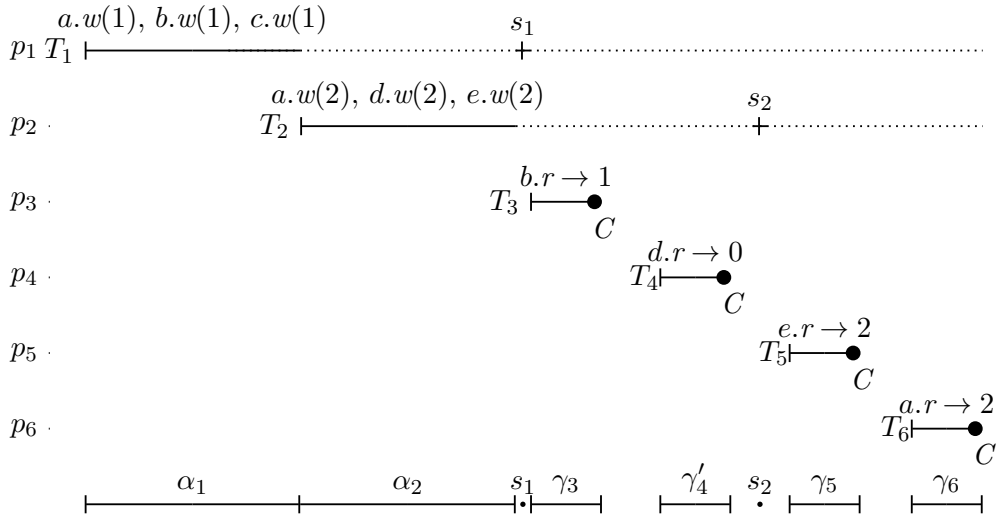

Figure 1: Execution γ in which T_6 reads 2 from a.

4 SI-DSTM

SI-DSTM is a simplified version of DSTM [10]. It maintains, for each update transaction T, a record with fields: (i) *Status*: stores the current status of T (takes values `Active`, `Committed`, or `Aborted`, initially `Active`), (ii) *pendingStatus*: records whether T should eventually abort (takes values `Active`, `Committed`, or `Aborted`, initially `Committed`), and (iii) *readList*: stores information about the data items read by T. As in DSTM, SI-DSTM maintains two records, `Locator` and `TMObject`, for each data item x and ensures a one-to-one correspondence between data items and `TMObjects`. It T wants to write a data item and the ownership of x is already held by T, then the new value is written in the *newObject* field of current x's locator. Otherwise, as in DSTM, cloning and indirection are employed: a new locator is created for x and its *transaction* field is initialized

to point to the transactional record of T. Then, T repeatedly tries to change the *start* field of the `TMObject` of x to point to this new locator. Before doing so, it writes the value `Aborted` in the *pendingStatus* field of the transactional record of the transaction that T found to be the holder of the ownership of x. Each transaction validates its read set to ensure that each data item in T's read list is still consistent. We remark that performing this validation only once at commit time is enough to ensure snapshot isolation.

In SI-DSTM, read-only transactions are invisible. Specifically, SI-DSTM does not maintain shared transactional records for read-only transactions. Each such transaction T maintains only a read list in its private memory space. Moreover, read-only transactions never cause other transactions to abort. From the transactional records of update transactions, a read-only transaction reads only the *status* field in order to find the current value of a data item that it wants to read and it is owned by any of these transactions; it does so in a way similar to that of DSTM. In order strict disjoint-access-parallelism to be ensured between a read-only transaction T and the update transactions, in SI-DSTM, `Locator` contains an additional field, called *pendingStatus*, in each transactional record. If a transaction T_1 perfroms a write operation to a data item x for which transaction T_2 holds the ownership, T_1 does not write the value `Abort` in the status field of T_2; it rather writes this value in *pendingStatus* to indicate that T_2 should eventually abort. At commit time, T_2 performs an exchange of *pendingStatus* and *status*. It then reads *status* again and returns `true` or `false` depending on the value of *status*. In this way, read-only transactions that read data items owned by T_2 contend only with T_2 and not with T_1 or other update transactions that write in the transactional record of T_2. Thus, strict disjoint access parallelism is ensured between a read-only transaction and any other transaction.

References

[1] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations (extended abstract). In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, PODC '97, pages 111–120, New York, NY, USA, 1997. ACM.

[2] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 69–78, New York, NY, USA, 2009. ACM.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995.

[4] D. Dice and N. Shavit. What really makes transactions faster? In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, TRANSACT'06, 2006.

[5] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Electron. Notes Theor. Comput. Sci.*, 259:245–261, Dec. 2009.

[6] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, PODC '12, pages 115–124, New York, NY, USA, 2012. ACM.

[7] F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, Sept. 1998.

[8] R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 304–313, New York, NY, USA, 2008. ACM.

[9] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, May 2005.

[10] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of ACM PODC'03*, pages 92–101. ACM, 2003.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, may 1993.

[12] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, PODC '94, pages 151–160, New York, NY, USA, 1994. ACM.

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

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

[15] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of ACM PODC'95*, pages 204–213, New York, NY, USA, 1995. ACM.