Brief Announcement: Towards a Fully-Articulated Pessimistic Distributed Transactional Memory *

Konrad Siek
Institute of Computing Science
Poznań University of Technology
Poznań, Poland
konrad.siek@cs.put.edu.pl

Paweł T. Wojciechowski
Institute of Computing Science
Poznań University of Technology
Poznań, Poland
pawel.t.wojciechowski@cs.put.edu.pl

ABSTRACT

Transactional memory, an approach aiming to replace cumbersome locking mechanisms in concurrent systems, has become a popular research topic. But due to problems posed by irrevocable operations (e.g., system calls), the viability of pessimistic concurrency control for transactional memory systems is being explored, in lieu of the more typical optimistic approach. However, in a distributed setting, where partial transaction failures may happen, the inability of pessimistic transactional memories to roll back is a major shortcoming. Therefore, this paper presents a novel transactional memory concurrency control algorithm that is both fully pessimistic and rollback-capable.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*

Keywords

Concurrency control; Software transactional memory

1. INTRODUCTION

Transactional Memory (TM) [8, 13] is an increasingly popular research topic and a promising way to reduce the effort overhead introduced by concurrent programming by using the *transaction* abstraction. This approach is also applied to distributed systems, although additional issues like partial failures need to be addressed there.

In TM emphasis is placed on optimistic concurrency control. There are variations, but generally speaking in this approach a transaction executes regardless of other transactions and performs validation only when it finishes executing (at commit-time). If two transactions try to access the same object, and one of them writes to it, they conflict and one of them aborts and restarts. When a transaction aborts, it should not change the system state, so aborting transactions must revert the objects they modified to a checkpoint. Alternatively, they work on local copies and merge them with the original object on a successful commit.

Unfortunately, there is a problem with irrevocable operations in the optimistic approach. Such operations as system calls, I/O operations, or network messages, once executed, cannot be canceled and so, cause aborted transactions to have a visible effect on the system. In a distributed context these operations are common. The problem was avoided by using irrevocable transactions that run sequentially, and so cannot abort [16], or providing multiple versions of transaction view for reads [1, 12]. In other cases, irrevocable operations are just forbidden in transactions (e.g., in Haskell).

A different approach, as suggested by [10] and our earlier work [17, 18], is to use fully-pessimistic concurrency control. This involves transactions waiting until they have permission to access shared objects. In effect, conflicting operations are postponed and transactions avoid forced aborts. And therefore, transactions naturally avoid the problems stemming from irrevocable operations.

However, distributed transactional memory, pessimistic or optimistic, must still support rollback, because it is possible for partial failures to occur in the system. More precisely, rollback is ideal for reverting the system to a consistent state as part of a recovery procedure. Moreover, rollback makes a TM more expressive. That is, there are situations where the programmer wants to abort transactions as part of program logic. If there is no rollback and the programmer implements an *ad hoc* stand-in within the transaction, this detract from readability and is inefficient due to extra network communication with objects. To make it efficient, the programmer has to make changes to the object code (i.e., allow object-local backup copies).

In this paper we propose the *Supremum Versioning Algorithm* (SVA) with rollback support, a novel algorithm for fully-pessimistic concurrency control aimed for distributed wide-area transactions. Given precise information on object use within transactions, SVA provides optimal scheduling of transaction operations comparable to manually-designed fine-grained locking. This is due to its ability to release objects before committing (after the object was used for the last time). The ability to use rollback allows SVA to react to failures and makes it more expressive and easier to use from the point of view of the programmer. SVA preserves opacity and strong progressiveness guarantees while supporting both rollbacks and the early release mechanism.

*This work was funded by NCN grant 2012/06/M/ST6 /00463.

SPAA'13, July 23–25 2013, Montréal, Québec, Canada.

$$
\begin{array}{lcl}
O & \subseteq & \mathsf{Objects} \\
T & \subseteq & \mathsf{Transactions} \\
L & \subseteq & \mathsf{Objects} \to \mathsf{Locks} \\
V_g, V_l, V_{lt}, V_c & \subseteq & \mathsf{Objects} \to \mathbb{N}_0 \\
V_p, C, V_r & \subseteq & \mathsf{Objects} \times \mathsf{Transactions} \to \mathbb{N}_0 \\
O_s & \subseteq & \mathsf{Objects} \times \mathsf{Transactions} \to \\
 & & \qquad \mathsf{ObjectData} \cup \{\texttt{null}\}
\end{array}
$$

Figure 1: SVA structures.

2. RELATED WORK

Several distributed TM systems were proposed (see e.g., [2, 5, 9, 19]). Most of them replicate a non-distributed TM on many nodes and guarantee that replicas are consistent. Their programming model is different from our distributed transactions. Other systems extend non-distributed TMs with a communication layer, e.g., DiSTM [9] extends [7] with distributed coherence protocols. HyFlow [15] uses a similar model to ours. However, these are optimistic TMs.

Distributed transactions are successfully used where requirements for strong consistency meet wide-area distribution, e.g., in Google's Percolator [11] and Spanner [4]. Percolator supports multi-row, ACID-compliant, pessimistic database transactions that guarantee snapshot isolation. A drawback in comparison to TM is that writes must follow reads. Spanner provides semi-relational replicated tables with general purpose distributed transactions. It uses real-time clocks and Paxos to guarantee consistent reads. Spanner defers commitment like SVA, but buffers writes and aborts on conflict. Irrevocable operations are banned in Spanner.

Matveev and Shavit [10] propose pessimistic non-distributed TM that runs transactions sequentially (as in [16]) but allows parallel read-only transactions (a plausible extension of SVA). Operations are synchronized by delaying writes of the write-set location (with busy waiting). This is done using version numbers of transactions. In contrast, SVA uses object versions for similar purposes, which enables early release. However, direct comparison is difficult, because [10] aims at non-distributed environments with fast access, while SVA assumes network communication with overheads.

In [3], the authors prove that in a TM with faulty processes, local progress (analogous to wait-freedom) and opacity cannot both be ensured. Faulty processes either crash or are *parasitic*—they run without ever attempting to commit or abort. This is also evident for SVA.

There is no rigorous performance comparison of pessimistic and optimistic (or hybrid) distributed TM. However, our previous work [19] can shed some light. We compared pessimistic state-machine–based and optimistic deferred-update replication schemes in a faulty system. We showed that neither scheme is clearly superior, but performance of the former is less dependent on workload and contention.

3. SVA WITH ROLLBACK

The *Supremum Versioning Algorithm* (SVA) is a pessimistic concurrency control algorithm with rollback support; it builds on our rollback-free variant in [18]. The *modus operandi* of SVA is that transactions receive a version number during initialization and use it to determine whether they can access a shared object or whether they must wait until another transaction finishes using that object. Finally a transaction commits, which means it releases its objects—other transactions can start using them and this transaction no longer will. Alternatively, the transaction can roll back (abort). This reverts the state of all shared objects as if the transaction never modified them and finally releases them.

```
1  proc @t start(t ∈ T, S ⊆ O → ℕ0 ∪ {ω}) ≜
2    for each o ∈ dom(S) according to ≺L do
3      @o lock L(o)
4    for each o ∈ dom(S) in parallel do
5      @o Vg[o ↦ Vg(o) + 1]
6      @o Vp[(o, t) ↦ Vg(o)]
7    for each o ∈ dom(S) according to ≺L do
8      @o unlock L(o)
9  proc @t call(t ∈ T, S ⊆ O → ℕ0 ∪ {ω}, o ∈ O) ≜
10   @o wait until Vp(o, t) − 1 = Vl(o)
11   @o checkpoint(t, o)
12   if Vr(o, t) ≠ Vc(o) then
13     @t rollback(t, S) and exit
14   @o call o
15   @o C[(o, t) ↦ C(o, t) + 1]
16   if C(o, t) = S(o) then
17     @o Vc[o ↦ Vp(o, t)]
18     @o Vl[o ↦ Vp(o, t)]
19 proc @t rollback(t ∈ T, S ⊆ O → ℕ0 ∪ {ω}) ≜
20   for each o ∈ dom(S) in parallel do
21     @o dismiss(t, o)
22     @o restore(t, o)
23 proc @t commit(t ∈ T, S ⊆ O → ℕ0 ∪ {ω}) ≜
24   for each o ∈ dom(S) in parallel do
25     @o dismiss(t, o)
26   if ∃o ∈ dom(S), Vr(o, t) > Vc(o) then
27     @t rollback(t, S) and exit
28   for each o ∈ dom(S) in parallel do
29     @o Os(o, t) ← null
30     @o Vlt[o ↦ Vp(o, t)]
31 proc @o checkpoint(t ∈ T, o ∈ O) ≜
32   if C(o, t) = 0 then
33     @o Os(o, t) ← copy o
34     @o Vr[(o, t) ↦ Vc(o)]
35 proc @o dismiss(t ∈ T, o ∈ O) ≜
36   @o wait until Vp(o, t) − 1 = Vlt(o)
37   if C(o, t) ≠ 0 ∧ Vr(o, t) = Vc(o) then
38     @o Vc[o ↦ Vp(o, t)]
39   if Vp(o, t) − 1 = Vl(o) then
40     @o Vl[o ↦ Vp(o, t)]
41 proc @o restore(t ∈ T, o ∈ O) ≜
42   if C(o, t) ≠ 0 ∧ Vr(o, t) < Vc(o) then
43     @o revert o ← Os(o, t)
44     @o Vc[o ↦ Vr(o, t)]
45   @o Os(o, t) ← null
46   @o Vlt[o ↦ Vp(o, t)]
```

Figure 2: SVA with rollback.

SVA supports an early release mechanism. That is, a transaction can release any objects after it used them for the last time (even before committing). This is possible because every transaction knows the upper bounds on the number of accesses to each object. (For our purposes this information comes from an oracle, but in practice, it may be extracted through static analysis [14] or typing [17].) However, while this development significantly improves the number of transactions executing simultaneously, it leads to a more complex rollback mechanism. A transaction must defer commit until the preceding transaction finishes, in case the current transaction uses objects that were released early and the preceding transaction rolls back. This is detailed below.

SVA Structures Before discussing the algorithm we describe the structures it uses. They are defined in Fig. 1. SVA works on sets of shared objects O and transactions T.

Since SVA works in a distributed system, objects are located and transactions spawn at arbitrary locations (or sites).

The basic premise of versioning algorithms is that counters are associated with transactions and used to allow or deny access by these transactions to shared objects (rather than only for recovery). SVA uses several version counters. *Private version counters* V_p uniquely define the version of a transaction with respect to an object. *Global version counters* show which transaction last started on a given object (this is used to initialize V_p). *Local version counters* V_l show which transaction can use a given object. *Local terminal version counters* V_{lt} show which transaction can commit or roll back a specific object. *Current version counters* V_c and *recovery version counters* V_r are used as a pair to detect if a preceding transaction rolled back an object that the current transaction is already using and to determine which transaction is responsible for reverting the object's state.

In order to detect the last use of an object, SVA requires that suprema (or infinity ω) on accesses S be given for each object used by a transaction. Then, *call counter* C is used to track actual accesses and to release objects early. In order be able to revert the state of objects SVA also uses a *stored object map* O_s, where transactions store copies of objects before modifying them. Also, SVA uses a map of locks L, one lock for each object during transaction start. These locks must always be used in order $\prec_L$ to prevent deadlocks.

The version counters and other structures are distributed among shared objects. Specifically, for any object o, values of V_p, V_l, V_{lt}, V_g, V_c, V_l, L, and O_s associated with o are located at o. Initially, all the locks are unlocked, all counters are set to zero, and the stored object map is empty.

SVA Transactions The life cycle of every SVA transaction begins with procedure start (we also refer to this part as initialization). Following that, a transaction may execute one or more accesses (calls) to a shared object. After any call or right after start a transaction may then either proceed to commit or rollback, which ends a transaction's life cycle. All procedures are shown in Fig. 2 and described below.

Note that accesses to shared objects can be interleaved with accesses to non-shared, transaction-local objects. However, those objects are only visible to the transaction they are local to, so they do not influence other transaction. So, we omit them for clarity. We also assume that transactions are executed in a single fresh dedicated thread. We also do not allow nested transactions and recurrency.

The pseudocode in Fig. 2 indicates where particular procedures and operations are located in the distributed system. If an operation is run on the host where object o is located, we mark it as @o. By analogy, @t means the operation is executed at the client running t. This gives a picture of network communication that transactions need to engage in.

The initialization of a transaction is shown in start at line 1. When a transaction starts it uses V_g to assign itself a unique version for each object it will access (V_p). This must be done atomically and in isolation, so these operations are guarded by locks—one lock l_o for each object used.

Objects are accessed via procedure call at line 9. Before accessing an object, the transaction waits for the preceding transaction to release it (line 10). When this happens, the transaction makes a backup copy to O_s using checkpoint (line 31)—a backup copy is made only before the first access to the object. If meanwhile no transaction modified the object by rolling back, it is accessed. Otherwise, the transaction also rolls back, because it would access inconsistent state. After accessing an object, transaction checks whether this was the last access using C. If so, the object is released early—i.e, V_l is set to the version of the transaction V_p.

A transaction can attempt to commit using the procedure at line 23. A commit first releases all objects used by the transaction (procedure dismiss at line 35). To do this, the transaction waits for the previous transaction to commit the object. Then, V_c and V_l are set to indicate the object is released and who is responsible for reverting it. This may not be necessary if the object was not used or was released early. Second, the transaction checks whether any transaction rolled back and modified its objects. If so, it is forced to roll back. Otherwise the transaction erases backup copies from O_s and completes the commit by setting V_{lt}.

If the programmer decides to roll back a transaction, or if a rollback is forced in an aforementioned situation, procedure rollback (line 19) is used. In that case, objects are released using dismiss (as described above). Afterwards, the transaction restores its objects using restore (line 41). If an object was accessed at least once, this procedure reverts it to a copy from O_s and sets V_c to show that this transaction is now responsible for rolling the object back (e.g., so a simultaneously aborting younger transaction won't override the copy). Finally, the transaction cleans up O_s and finishes rolling back by setting V_{lt}.

4. PROPERTIES AND CORRECTNESS

In this section, we show that SVA guarantees safety (opacity) and liveness (strong progressiveness). The property of opacity is defined as follows (after [6]):

DEFINITION 1. *A finite TM history H is final-state opaque if there exists a sequential TM history S equivalent to any completion of H, such that (1) S preserves the real-time order of H, (2) every transaction $t \in T$ in S is legal in S.*

DEFINITION 2. *A TM object o is opaque if, and only if, every finite history of o is final-state opaque.*

Before defining strong progressiveness, it is necessary to define sets $CTrans(H)$, $CObj_H(t_i)$, and $CObj_H(Q)$. $CTrans(H)$ is a set of all subsets Q of all transactions in a history H, such that Q is not empty and no transaction in Q conflicts with any transaction not in Q. $CObj_H(t_i)$ is a set of shared objects, such that object o is included in the set if there exists a transaction t_j $(i \neq j)$ in history H that conflicts with transaction t_i on shared object o. Given a set of transactions Q, $CObj_H(Q)$ is a union $\bigcup_{t_k \in Q} CObj_H(t_k)$.

Given this, strong progressiveness is defined as follows:

DEFINITION 3. *A TM history H is strongly progressive if, for every set $Q \in CTrans(H)$ such that $|CObj_H(Q)| \leq 1$, at least one transaction in Q is not forcibly aborted in H.*

DEFINITION 4. *A TM object o is strongly progressive if every history of o is strongly progressive.*

4.1 Opacity

THEOREM 1. *A finite SVA history is final-state opaque.*

PROOF. Since all SVA transactions access each object o only when the guard condition $V_p(o,t) - 1 = V_l(o)$ (line 10

in Fig. 2) they have exclusive access to o and they do so according to total order $\prec_{T'}$.

An SVA transaction t can access o when a previous transaction t' releases o early. In that case the upper bound on its use by t' was reached, so the state of o is the same as if t' committed. Also, t will wait on condition $V_p(o,t) - 1 = V_{lt}(o)$ (line 36 in Fig. 2) until t' finishes and if t' aborts, t will also abort. So no SVA transaction sees an inconsistent state of the system, nor does any committing transaction see changes from an aborting transaction.

Let H' be a completion of any SVA history H such that every live or commit-pending transaction in H is aborted. Since accesses to each o are totally ordered, this imposes a partial order $\preceq_{H'}$ on H'. Given this, we can construct a sequential witness history S equivalent to completion H' ($H'|t = S|t$ for every t), where there is a total order $\prec_S$ on S, such that $\preceq_{H'} \subseteq \prec_S$. Thus, we can construct an equivalent sequential history S that preserves the real-time order of H'.

Since committing transactions do no see the effects of aborted transactions and have exclusive access to objects they use for the duration of their execution, and since live transactions do not see an inconsistent state of the system, then for any H', transactions will behave as if they were executed sequentially, so transactions in the sequential history S will conform to a sequential specification of the shared objects. Therefore, every transaction will be legal in S.

Since there exists a sequential history S equivalent to H that preserves the real time order of H and every transaction t in S is legal in S. Therefore H is final-state opaque. □

THEOREM 2. *SVA is opaque.*

PROOF. By Definition 1 and Definition 2 any transactional object o is opaque if, and only if, every finite history of o is final-state opaque. Since the latter follows from Theorem 1, then it is true that SVA objects are opaque, and therefore SVA is opaque. □

4.2 Strong progressiveness

THEOREM 3. *Every SVA history is strongly progressive.*

PROOF. Note also that an SVA transaction t may only be forcibly aborted if for some object o, $V_r(o,t) \neq V_c(o)$ or $V_r(o,t) > V_c(o)$ (line 13 and line 27 in Fig. 2). This happens if there is another transaction t' that also accesses object o, t' accesses o prior to t and releases it early so that t can access it, and finally t' aborts.

If there is a set of transactions $Q \in CTrans(H)$ such that $|CObj_H(Q)| = 0$ then no two transactions in Q attempt to access the same object. Therefore no transaction can force another to abort. If there is a set of transactions $Q \in CTrans(H)$ such that $|CObj_H(Q)| = 1$ then all transactions in Q share exactly one remote object o. Since all transactions in Q are ordered using a total order $\prec_Q$ it then follows that there is some transaction $t \in Q$ such that t is not preceded by any transaction, and therefore no transaction can cause it to abort. Therefore, there exists in Q a transaction that cannot be forcibly aborted.

Since transactions in $Q \in CTrans(H)$ such that $|CObj_H(Q)| = 0$ cannot be forcibly aborted and since there exists at least one transaction t in $Q \in CTrans(H)$ where $|CObj_H(Q)| = 1$ such that t cannot be aborted, then it is true that for any set Q where $|CObj_H(Q)| \leq 1$ some transaction is not forcibly aborted in H. Therefore every SVA history H is strongly progressive in accordance with Definition 3. □

THEOREM 4. *SVA is strongly progressive.*

PROOF. Since Theorem 3 shows that every SVA history is strongly progressive, it follows from Definition 4 that SVA objects are strongly progressive. □

5. REFERENCES

[1] H. Attiya and E. Hillel. Single-version STMs can be multi-version permissive. In *Proc. ICDCD'11*, 2011.

[2] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *Proc. PPoPP'08*, Feb. 2008.

[3] V. Bushkov, R. Guerraoui, and M. Kapałka. On the liveness of transactional memory. In *Proc. PODC'12*, July 2012.

[4] J. C. Corbett and et al. Spanner: Google's globally-distributed database. In *Proc. OSDI'12*, 2012.

[5] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *Proc. PRDC'09*, Nov. 2009.

[6] R. Guerraoui and M. Kapałka. *Principles of Transactional Memory*. Morgan & Claypool, 2010.

[7] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *Proc. OOPSLA'06*, Oct. 2006.

[8] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. ISCA'93*, May 1993.

[9] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. C. Kirkham, and I. Watson. DiSTM: A software transactional memory framework for clusters. In *Proc. ICPP'08*, Sept. 2008.

[10] A. Matveev and N. Shavit. Towards a fully pessimistic STM model. In *Proc. TRANSACT'12*, Aug. 2012.

[11] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proc. OSDI'10*, Oct. 2010.

[12] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *Proc. PODC'10*, 2010.

[13] N. Shavit and D. Touitou. Software transactional memory. In *Proc. PODC'95*, Aug. 1995.

[14] K. Siek and P. T. Wojciechowski. A formal design of a tool for static analysis of upper bounds on object calls in Java. In *Proc. FMICS'12*, LNCS 7437, Aug. 2012.

[15] A. Turcu and B. Ravindran. On open nesting in distributed transactional memory. In *Proc. SYSTOR'12*, June 2012.

[16] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *Proc. SPAA'08*, June 2008.

[17] P. T. Wojciechowski. Isolation-only transactions by typing and versioning. In *Proc. PPDP '05*, July 2005.

[18] P. T. Wojciechowski. *Language Design for Atomicity, Declarative Synchronization, and Dynamic Update in Communicating Systems*. Poznań University of Technology Press, 2007.

[19] P. T. Wojciechowski, T. Kobus, and M. Kokociński. Model-driven comparison of state-machine-based and deferred-update replication schemes. In *Proc. SRDS'12*, Oct. 2012.