Zen and the Art of Concurrency Control: An Exploration of TM Safety Property Space with Early Release in Mind

Konrad Siek Paweł T. Wojciechowski

Poznań University of Technology
Institute of Computing Science
konrad.siek@cs.put.edu.pl pawel.t.wojciechowski@cs.put.edu.pl

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming

Keywords Concurrency control; Software transactional memory; Opacity; Virtual World Consistency

1. Introduction

Transaction Memory (TM) [9, 12] is a concurrency control abstraction that allows the programmer to specify blocks of code to be executed atomically (or with particular guarantees). However, since transactional code can contain just about any operation, rather than just reads and writes of TM's database predecessors, a greater attention must be payed to the state of shared variables at any given time. E.g., if a database transaction reads a stale value, it must simply abort and retry, and no harm is done. Whereas, if a TM transaction reads a stale value it may execute dangerous operations, like attempt to divide by zero, access an illegal memory address, or enter an infinite loop. Thus strong safety properties are considered important in TM, such as *opacity* [6] which regulate what values can be read, even by transactions that abort. In comparison to these, standard database consistency conditions such as *serializability* [11] are relatively weak.

However, such strong properties preclude (or virtually preclude) using early release as a technique for optimizing TM (see [5, 13]). Early release is a mechanism by which transactions forgo exclusive access to shared variables in return for increased parallelism of operations between transactions. In order to maintain the correctness of the global state, transactions which read an inconsistent value are aborted, but this does not prevent them from viewing at least some inconsistent data. Hence, they cannot satisfy standard properties like opacity, etc. On the other hand, serializability does not describe the TM's guarantees adequately, since the TM can preclude some, if not all, inconsistent views, and therefore be much more safe and practical than a serializable TM.

Hence, in this paper we explore the TM safety property space: serializability, opacity *virtual world consistency* [10], and the *TMS* family [4], and consider whether they support early release and to what extent. We do this in Section 3, but define our terms beforehand in Section 2. In the latter part of Section 3, we take a look at some database consistency properties designed for transactional processing in databases. We consider them pertinent, since database transactions have many similarities to TM. Finally, in Section 4, we specify how serializability can be combined with some database properties to create a broader spectrum of useful early release supporting TM safety properties. We fill the remaining gap by proposing *Last-use consistency*, a consistency property that excludes those inconsistent views which are the most troublesome, but allows others.

2. Definitions

First, let us introduce several well-known TM terms (after [7]). Consider a read operation op_r in transaction T_i that reads value v from some variable x, and a write operation op_w in T_j that precedes op_r in H and writes v to x. Operation op_r is *local* if op_w is part of the same transaction as op_r and *non-local* otherwise. Transaction T_i is *committed* if T_i begins its commit procedure and finishes successfully. If the commit procedure is not yet completed (so T_i can still be aborted), then T_i is *commit-pending*. If transaction T_i is neither committed nor commit-pending, it is *live*.

We also introduce a concept of *commit-pending equivalence*. A commit-pending–equivalent transaction with respect to x is a transaction that is live, but it executed its last possible write on x and will perform no further writes on x before it either commits or aborts. This is defined formally as follows:

Definition 1 (Commit-pending Equivalence). *Live transaction T_i in H is commit-pending–equivalent with respect to x if there exists read or write operation op on variable x in $H|T_i$, such that for any history H_c for which H is a prefix ($H_c = H \cdot H'$) operation op is the last read or write operation on x in $H_c|T_i$ (i.e., there is no write on x in $H'|T_i$).*

Early release pertains to any situation where conflicting transactions execute partially in parallel and access the same object. We assume unique writes for convenience, but without loss of generality (see [7]). We define the concept of early release as follows:

Definition 2 (Early Release). *Transaction T_i releases x early in H iff there is some prefix H' of H, such that T_i is live in H' and there exists T_j in H' such that there is a non-local read operation op_j in $T_j|H'$ reading v from x and a preceding write operation op_i in $T_i|H'$ writing x to v.*

We also distinguish a more specific case where one transaction accesses another transaction's variables only once the transaction no longer uses them (but is still live). We define this as follows:

Definition 3 (Last-use Release). *Transaction T_i releases x after last-use in H iff T_i releases x early in H and T_i is commit-pending equivalent wrt x.*

Finally, let us define more formally what it means for a safety property to support early release of one form or another. Let $\mathbb{P}$ be a safety property, and let H be any history that satisfies $\mathbb{P}$. Then:

Definition 4 (Early Release Support). *$\mathbb{P}$ supports early release iff there exists some transaction T_i in H satisfying $\mathbb{P}$ such that for some variable x, T_i releases x early in H.*

Definition 5 (Last-use Release Only Support). *$\mathbb{P}$ supports last-use release only iff $\mathbb{P}$ supports early release, and for any T_i in H*

satisfying $\mathbb{P}$, if T_i releases any x early, then T_i releases x after last-use.

3. Early Release Support and Safety Properties

Let us discuss whether particular safety properties allow for early release. The properties under consideration are some TM safety properties as well as database properties pertaining to transactional processing.

3.1 Serializability

First, let us consider serializability, which can be considered a baseline TM safety property. It is defined in [11] in three variants: conflict serializability, view serializability, and final-state serializability. We follow a more general version of serializability defined in [14], which we summarize as follows:

Definition 6 (Serializability)**.** *History H is serializable iff there exists some linear extension (sequential witness history) H_S such that H_S only contains legal transactions.*

This definition does not preclude early release, as long as illegal transactions are aborted. Serializability also does not limit early release to releasing after last use. Thus, we stipulate the following.

Lemma 1. *Serializability supports early release.*

Proof sketch. Let H be a transactional history as follows:

$$\begin{array}{lll} T_i & [\![\, \mathrm{w}(x)0,\ \mathrm{w}(x)1 \,]\!] & \\ T_j & \quad [\![\, \mathrm{r}(x)0,\ \circlearrowleft \ldots T'_j & [\![\, \mathrm{r}(x)1, \mathrm{w}(x)2 \,]\!] \end{array}$$

Here, T_i performs two writes to x and T_j reads and writes to x. Also, T_j reads a value of x that T_i later overwrites, so T_j rolls back (denoted $\circlearrowleft$) and retries later. The linear extension of H is $\hat{S} = T_i [\![\mathrm{w}(x)0, \mathrm{w}(x)1]\!], T_j [\![\mathrm{r}(x)1, \mathrm{w}(x)2]\!]$, wherein all transactions are legal. Thus H is serializable. Since, by Def. 2, T_i releases early in H, then, by Def. 4, serializability supports early release. □

Lemma 2. *Serializability does not support last-use release only.*

Proof sketch. Let H be a history as above. Transaction T_i writes to x after T_j reads x, so by Def. 1, T_i is not commit-pending–equivalent wrt x when T_j reads x. Since this is contrary to Def. 3, T_i does not release x after last use. This, in turn, is contrary to Def. 5, so serializability does not support last-use release only. □

3.2 Opacity

Opacity [6, 7] can be considered the standard TM safety property. It requires serializability, preservation of real-time order, and consistency. Only the first and latter-most components are pertinent to our current exploration. We deal with serializability in Lemmas 1–2. Consistency is defined in [7] as follows (in a brief form):

Definition 7 (Consistency)**.** *History H is consistent, if for every read operation $op_r = \mathrm{r}(x)v$ on variable x returning value v in subhistory $H|T_i$, it is true that:*

a) If op_r is local then the latest write operation on x preceding op_r writes v to x.

b) If op_r is non-local then either $v = 0$ or there is a non-local write operation on x writing v in $H|T_k$ ($T_k \neq T_i$) where T_k is committed or commit-pending.

This definition of consistency precludes any use of early release whatsoever, so the following is true about opacity:

Lemma 3. *Opacity does not support early release.*

Proof sketch. If opacity is to support early release, then by Def. 4, there must exist some T_i in some opaque H, such that T_i releases some x early. I.e., by Def. 2, some T_j reads from T_i while T_i is live. This implies that there is some non-local operation *op* in $H|T_j$ that reads a value of some variable written by another operation in $H|T_i$. However, T_i is live, which is at odds with Def. 7b, and since consistency is required for opacity, it is impossible for there to exist such T_i. Hence opacity precludes early release support. □

Lemma 4. *Opacity does not support last-use release only.*

Proof sketch. Since last-use release support requires early-release support (Def. 5), then the lemma follows from Lemma 3. □

3.3 Virtual World Consistency

Since the requirements of opacity, while very important in the context of TM's ability to execute any operation transactionally, can often be excessively stringent. On the other hand serializability is considered too weak for many TM applications. Thus, *Virtual world consistency* (VWC) was introduced, which is a TM consistency condition defined as follows (from [10]):

Definition 8. *History H meets VWC if all committed transactions are serializable and preserve real-time order, and for all aborted transactions there exists a linear extension of its causal past that is legal.*

While weaker than opacity, this property nevertheless precludes use of early release, as follows.

Lemma 5. *VWC supports early release.*

Proof sketch. Let H be a transactional history as follows:

$$\begin{array}{ll} T_i & [\![\, \mathrm{r}(x)0, \mathrm{w}(x)1, \mathrm{r}(y)0 \,]\!] \\ T_j & \qquad\qquad\quad [\![\, \mathrm{r}(x)1 \,]\!] \end{array}$$

Here, T_i performs two operations on x and one on y, while T_j reads x. The linear extension of H is $\hat{S} = T_i [\![\mathrm{r}(x)0, \mathrm{w}(x)1, \mathrm{r}(y)0]\!]$, $T_j [\![\mathrm{r}(x)1]\!]$, wherein all transactions are legal. Thus H is VWC. Since, by Def. 2, T_i releases early in H, then, by Def. 4, VWC supports early release. □

Lemma 6. *VWC supports last-use release only.*

Proof sketch. Since VWC requires that aborting transactions view a legal causal past, then if a transaction reading x is aborted, it must read a legal (i.e. "not overwritten") value of x. Thus, let us consider some history H where some T_i releases x early, and some T_j reads x from T_i.

a) If T_i writes to x after releasing it, and T_j commits, then T_j is not legal, and therefore H does not satisfy VWC.

b) If T_i writes to x after releasing it, and T_j aborts, then the causal past of T_j contains T_i, and T_j reads an illegal (stale) value of x from T_i, so H does not satisfy VWC.

Therefore, any history H containing T_i, such that T_i releases x early and modifies it after release does not satisfy VWC. Therefore in any VWC history H containing some T_i, such that T_i releases x early, T_i releases x after last use. Thus, by Def. 5, the lemma holds. □

While VWC allows supports early release, there are severe limitations to this capability. I.e., VWC does not allow a transaction that released early to subsequently abort for any reason. First, let us remind that by its definition in [10], given transaction T_i, causal past $C(T_i)$ is legal, if for every $T_j \in C(T_i)$, s.t. $i \neq j$, T_j is committed. Then let us state as follows:

Lemma 7. *Given a VWC history H, T_i releases x early, then T_i cannot abort.*

Proof Sketch. By contradiction, let us assume that T_i eventually aborts. By Def. 2, there is some T_j in H that reads from T_i. If T_i eventually aborts, then T_j reads from an aborted transaction.

a) If T_j eventually aborts, then its causal past contains two aborted transactions (T_i and T_j) and is, therefore, illegal. Hence H does not satisfy VWC, which is a contradiction.

b) If T_j eventually commits, then the sequential witness history is also illegal. Hence H does not satisfy VWC, which is a contradiction.

Therefore, if T_i eventually aborts, H does not satisfy VWC, which is a contradiction. Thus, since a VWC history cannot contain an abortable transaction that releases a variable early, it cannot contain transaction T_i. □

We submit, that requiring a transaction to never abort after performing a release is impractical, since TM systems cannot predict whether any particular transaction eventually commits or aborts. An exception to this may be found in systems making special provisions to ensure irrevocable transactions eventually commit (see e.g. [15]), but, case in point, these take drastic measures to ensure that e.g. a single transaction is present in the system at one time. In the general case, however, the requirement is too strict.

3.4 TMS1

In [4] the authors argue that some contexts, such as sharing variables between transactional and nontransactional code, require additional safety properties. Thus, they propose and rigorously define two consistency conditions for TM: *TMS1* and *TMS2*. We only examine TMS1 here, but since TMS2 is strictly stronger than TMS1, our conclusions will equally apply to TMS2.

TMS1 follows a set of design principles including a requirement for observing consistent behavior or partial effects of transactions. The principle is reflected in the definition of the TMS1 automaton, and we paraphrase the relevant parts of the condition for the correctness of an operation's response in the following observation (see the definition of *validResp* for TMS1 in [4]).

Observation 1 (Valid Response)**.** *For operation op to return in some subhistory $H|T_i$, there must exists some set of transactions S that follow real-time order, justify the legality of op, and for any $T_j inS$ it is true that either:*

a) T_j precedes T_i in real-time order and T_j is committed, or

b) T_j is committed or commit-pending.

Then it is straightforward to see that:

Lemma 8. *TMS1 does not support early release.*

Proof sketch. If TMS1 is to support early release, then by Def. 4, there must exist some T_i in some H respecting TMS1, such that T_i releases some x early. I.e., by Def. 2, some T_j reads from T_i while T_i is live. Since live transactions cannot be committed or commit-pending, this is directly contradicted by Observation 1b. Thus, it is impossible for there to exist such T_i. Hence, TMS1 does not support early release. □

Lemma 9. *TMS1 does not support last-use release only.*

Proof sketch. By analogy to the proof of Lemma 4. □

3.5 Database Properties

We follow the discussion of TM safety properties with brief foray into database properties that deal with transaction consistency. Given that TM properties tend not to be very helpful when describing the behavior of early release, these consistency properties may be used to supplement that.

Recoverability *Recoverability* is a database property defined as below (following [8]):

Definition 9. *History H is recoverable if for any $T_i, T_j \in H$ s.t. T_j reads from T_i, T_i commits in H before T_j.*

Recoverability does not make requirements about values read by transactions, so it necessarily supports both early release and last-use release. However recoverability imposes an order on commits and aborts, and as such can be very useful in conjunction with e.g. serializability to limit the set of acceptable histories.

Avoiding Cascading Aborts *Avoiding cascading aborts* (ACA) [2] is a database property defined as:

Definition 10. *ACA holds for history H, if for any $T_i, T_j \in H$ s.t. T_j reads from T_i, T_i commits before the read.*

As with recoverability, ACA restricts reading from live transactions but allows writing after live transactions. Therefore ACA removes some early release scenarios, while still allowing it in some cases. As practice shows, (see e.g. [5, 13]), transactions can read inconsistent data as a result of early release, and may need to be aborted, which ACA prevents. Therefore, ACA may be useful in the context of early release in TM.

Strictness *Strictness* [2] is a database property defined as:

Definition 11. *History H is strict iff for any $T_i, T_j \in H$ and given any operation $op_i = \mathrm{r}(x)v$ or $\mathrm{w}(x)v'$ in $H|T_i$, and any operation $op_j = \mathrm{w}(x)v$ in $H|T_j$, if op_i follows op_j, then T_j commits or aborts before op_i.*

The definition unequivocally states that a transaction can read from or write after another transaction, the former transaction must be committed or aborted before this takes place, so strictness precludes early release and last-use release.

Rigorousness *Rigorousness* is defined (following [3]) as:

Definition 12. *History H is rigorous if it is strict and for any $T_i, T_j \in H$ such that T_j writes to variable x, i.e., $op_j = \mathrm{w}(x)v \in H|T_j$ after T_i reads x, then T_i commits or aborts before op_j.*

Since [1] demonstrates that rigorous histories are opaque, and since we show at the beginning of this section that opaque histories support neither early release nor last-use release, then neither does rigorousness.

4. Implications

There is a lack of sufficient spectrum of TM safety properties that would describe behavior of TM with early release adequately. Currently, TM system that uses early release can satisfy serializability, which is relatively weak, as far as TM safety properties are concerned. On the other hand, stronger properties include leave only opacity and TMS1, which disallow early release altogether, or VWC, which requires an impractical assumption that transactions releasing early not abort. However, a practical TM system that is serializable and limits early release to accept only a part of all histories with early release, but not enough to achieve either opacity, TMS1, or VWC lacks description.

4.1 Complementing Consistency with Database Properties

One way to attempt to mitigate this problem is to use the aforementioned database transactional processing properties to supplement the spectrum of safety properties. Specifically, two of the database properties support early release: recoverability and ACA, with the former being strictly weaker than the latter (see [1]). Thus, if these properties are used in conjunction with serializability, it is possible to form a spectrum of early-release–friendly TM consistency properties: serializability, *serializability + recoverability* (S+R), and *serializability + ACA* (S+ACA).

S+R limits the set of accepted histories to those which maintain the order of commits in agreement with the order in which transactions access variables. This prevents a situation such as the following, which is serializable but not recoverable:

$$\begin{array}{ll} T_i & \llbracket\, \mathrm{w}(x)0,\quad \mathrm{w}(x)1\, \rrbracket \\ T_j & \qquad\qquad \llbracket\, \mathrm{r}(x)0 \rrbracket \end{array}$$

However S+R is still a fairly weak property, especially in the context of early release, as it does not regulate inconsistent views. We believe that both DATM [5] and SVA [13] satisfy S+R, because they both are serializable and both order their commits following the order of the reads-from relation.

S+ACA, on the other hand, restricts the use of forced rollbacks, so that the following scenario is not S+ACA:

$$\begin{array}{ll} T_i & \llbracket\, \mathrm{w}(x)0,\ \ \mathrm{w}(x)1\, \rrbracket \\ T_j & \qquad\qquad \llbracket\, \mathrm{r}(x)0,\ \ \circlearrowleft \ldots T_j' \ \ \llbracket\, \mathrm{r}(x)1, \mathrm{w}(x)2\, \rrbracket \end{array}$$

Indeed, all $W \rightarrow R$ dependences between transactions are removed in the case of transactions that release early and only $W \rightarrow W$ dependences are retained. Since $W \rightarrow R$, can be considered the typical case in case of early release, we consider S+ACA to be very restrictive and difficult to implement. On the other hand, we conjecture S+ACA prevents all inconsistent views in a TM with early release. Thus, we believe that neither DATM nor SVA are S+ACA, since they both allow inconsistent views, and, indeed, both permit cascading aborts to occur.

4.2 Allowing Inconsistent Views

Thus, S+R does not eliminate inconsistent views and S+ACA precludes them. However we still lack properties which allow inconsistent views, but limit them only to a specific subset. We submit that a good half-way point between precluding inconsistent views and allowing them, would be only to allow from the subset of all transactions that release variables early, only those that do so after last use of each of the variables in question. In this way, no variable would see a value of a variable that would be overwritten (later modified) by a preceding transaction. This precaution precludes most of the problems with inconsistent views, as a transaction that reads a value of a variable that was released early will not expect the value to suddenly change, although the transaction may expect the preceding transaction itself to abort for some reason and invalidate the variable as a result.

Let as then propose a consistency condition, which here we will refer to as *last-use consistency* (LUC). Let $\mathbb{T}^H_{er}$ be a subset of all transactions in history H, such that $T_i \in \mathbb{T}^H_{er}$ iff for some x, T_i releases x early (Def. 2). Then, let $\mathbb{T}^H_{lu}$ be a subset of $\mathbb{T}^H_{er}$, such that $T_i \in \mathbb{T}^H_{lu}$ iff for any x that T_i releases early, T_i releases x after last use (Def. 3). Then LUC can be defined simply as follows:

Definition 13 (Last-use Consistency)**.** *History H satisfies LUC, if $\mathbb{T}^H_{lu} = \mathbb{T}^H_{er}$.*

Then, we can fill the spectrum of properties for early release with *serializability + recoverability + LUC* (S+R+LUC)—we add recoverability so that the property is strictly stronger than S+R.

The following history is precluded by S+R+LUC due, since T_i does not release early after last use, and therefore T_j reads an inconsistent value of x and is forced to abort.

$$\begin{array}{ll} T_i & \llbracket\, \mathrm{w}(x)0,\ \ \mathrm{w}(x)1\, \rrbracket \\ T_j & \qquad\qquad \llbracket\, \mathrm{r}(x)0,\ \ \circlearrowleft \ldots T_j' \ \ \llbracket\, \mathrm{r}(x)1, \mathrm{w}(x)2\, \rrbracket \end{array}$$

On the other hand, the following history is allowed by S+R+LUC, since T_i releases after last use, and despite that fact, that T_j reads an inconsistent value of x from the aborted (perhaps programmatically) T_i:

$$\begin{array}{ll} T_i & \llbracket\, \mathrm{w}(x)0,\ \ \mathrm{w}(x)1\ \circlearrowleft \\ T_j & \qquad\qquad\qquad\qquad \llbracket\, \mathrm{r}(x)0,\ \ \circlearrowleft \ldots T_j' \ \ \llbracket\, \mathrm{r}(x)1, \mathrm{w}(x)2\, \rrbracket \end{array}$$

Acknowledgments

The project was funded from National Science Centre funds granted by decision No. DEC-2012/07/B/ST6/01230.

References

[1] H. Attiya and S. Hans. Transactions are Back—but How Different They Are? In *Proceedings of TRANSACT'14: the 7th ACM SIGPLAN Workshop on Transactional Computing*, Feb. 2014.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

[3] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, 17, Sept. 1991.

[4] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25:769–799, Sept. 2013.

[5] H. e. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing Conflicting Transactions in an STM. In *Proceedings of PPoPP'09: the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2009.

[6] R. Guerraoui and M. Kapałka. On the Correctness of Transactional Memory. In *Proceedings of PPoPP'08: the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2008.

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

[8] V. Hadzilacos. A theory of reliability in database systems. *Journal of the ACM*, 35, Jan. 1988.

[9] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of ISCA'93: the 20th International Symposium on Computer Architecture*, May 1993.

[10] D. Imbs, J. R. de Mendivil, and M. Raynal. On the Consistency Conditions or Transactional Memories. Technical Report 1917, IRISA, Dec. 2008.

[11] C. H. Papadimitrou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, 1979.

[12] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of PODC'95: the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Aug. 1995.

[13] K. Siek and P. T. Wojciechowski. Brief announcement: Towards a Fully-Articulated Pessimistic Distributed Transactional Memory. In *Proceedings of SPAA'13: the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, July 2013.

[14] W. E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, Apr. 1989.

[15] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and their Applications. In *Proceedings of SPAA'08: the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, June 2008.