View Transactions and the Relaxation of Consistency Checks in Software Transactional Memory

Yehuda Afek Adam Morrison Moran Tzafrir

School of Computer Science
Tel Aviv University

Abstract

We present *view transactions*, a model for relaxed consistency checks in software transactional memory (STM). View transactions always operate on a consistent snapshot of memory but may commit in a different snapshot. They are therefore simpler to reason about, provide opacity and maintain composability. In addition, view transactions avoid many of the overheads associated with previous approaches for relaxing consistency checks.

1 Introduction

Transactional memory implementations are prone to *false aborts*, aborts of transactions that could be correctly serialized at the program semantics level, but that still conflict at the low level of memory reads and writes. For example, consider the *set* abstract data type[1] implemented by a sorted linked list. A transaction T_i executing `insert(205)` that is about to link a new node after a node with key 203 does not need to abort if another transaction T_d concurrently deletes a node with key 17, even though that deletion invalidates T_i's snapshot of the list. In addition, STMs need to constantly perform *consistency checks* to guarantee the serializability of all low level memory operations, and these checks incur a considerable performance hit.

To solve these issues, researchers have proposed *relaxing STM consistency checks* in different ways so that transactions that conflict at the memory level can still commit if they do not conflict at the program semantics level. Two prominent examples of such proposals are the *early release* model of [5] and the recent *elastic transactions* model [3]. In both proposals, it is up to the programmer to decide *when* to use relaxed checks, and making this decision requires reasoning about whether correct semantics of the program will be preserved under the relaxed checks. Thus, in contrast to the standard TM model, a programmer mistake may lead to a non-serializable incorrect program.

We are therefore interested in making transactions with relaxed consistency checks easier to use and reason about, to decrease the chance of programmers making mistakes when applying them. In this work we propose *view transactions*, a new model for relaxed consistency checks. Unlike the previous proposals, in which a transaction with relaxed consistency checks did not work on a consistent snapshot of memory and therefore may have observed updates by concurrent transactions, a view transaction *always operates on a consistent state (snapshot) of memory*. The relaxation of consistency checks is that a view transaction may commit in a different snapshot than the one it worked on. A sufficient condition for program correctness with view transactions is that the commit-time snapshot must be such that had the transaction operated on it, its externally visible actions would be the same. The programmer must therefore identify the transaction's *critical view* — a subset of the run-time snapshot that needs to be validated at commit-time. We introduce *view pointers*, an idea for easing the specification of the critical view. View transactions improve on previous relaxed consistency approaches on two fronts, *performance* and *usability*.

2 Benefit of view transactions

View transactions combine the following properties, that are not all present together in the previous relaxed transactional consistency models. The discussion below is summarized in Table 2.

Simple reasoning due to opacity. Because view transactions work on a snapshot, reasoning about them is closer to traditional *sequential* reasoning — the programmer need not worry about handling concurrent

[1] The *set* abstract data type maintains a set of items and supports the operations `contains()`, `insert()` and `remove()` on the set.

	General?	Transaction sees consistent state?	Composable?	Performance
Early release [5]	yes	can be adapted at the cost of performance (Section 3)	yes	high only if not working on snapshot
Elastic transactions [3]	consistency checks dictated by the model and not user controlled	no	only sometimes	high
View transactions	yes	yes	yes	high

Table 1: Properties of relaxed transactional consistency models.

updates. To date, relaxed consistency checks were used mainly in data structures such as lists and trees, where observing inconsistencies seems not to lead to serious errors. [2] But to facilitate wider use of relaxed consistency checks, programmers should be relieved from worrying about these inconsistencies.

Generality. View transactions are flexible enough to implement both the early release and the elastic transactional models (while still operating on a snapshot), i.e., view transactions can be made to generate a subset of the executions possible with these models. Therefore, proving correctness (even in these models) should be easier when reasoning about the corresponding "emulating" view transactions, since the executions that arise with view transactions are a subset of all possible executions of the "emulated" models.

Composability. The ability to *compose* view transactions. For example, the critical view of a list `contains()` that fails consists of the nodes where the traversal stopped. This ensures that a subsequent insertion of the item will be detected. Identifying critical views so that composability can be achieved requires care on the part of the programmer, but it is at least *possible* while still maintaining high performance. In elastic transactions composability is not always possible, and in early release performance suffers if it is adapted to work on a snapshot (see next section). Composability allowed us to adapt the STAMP benchmark `vacation` to view transactions almost trivially.

Performance. We defer analysis of view transactions' performance to the next section where we describe our implementation. The end conclusion, however, is that view transactions are comparable or outperform previous proposals on the benchmarks we evaluated. Thus, the original goal of achieving good performance is not harmed by the additional features of view transactions.

3 View transactions implementation

In contrast to DSTM [5], which relies on continuously validating the read set to maintain a snapshot and can therefore violate opacity when locations are removed from the read set using early release, modern STMs maintain a snapshot using location versions and a global clock [2]. It is therefore seemingly easy to adapt early release to a modern STM and obtain most of the conceptual benefits of view transactions. However, *many false aborts can still occur due to the version check* when a transaction finds that a location contains a value that is not consistent with its snapshot, forcing the STM to abort it. As we discuss below, this effect can completely destroy the performance benefit from relaxed consistency checks.

View transactions overcome this problem using *multiversions* [1] (a technique originally used in database systems) in which it is possible to access older versions of a location. Thus, a view transaction may observe *older* values (that are nevertheless consistent with its run-time snapshot) and thereby avoid false aborts, as these values may not need to be valid in the commit-time snapshot. Multiversions therefore avoid forcing the relaxed transactions to deal with inconsistent data, as happens in previous approaches.

To exploit multiversions, view transaction use a new *light read* primitive, which returns a value consistent

[2] Even so, reasoning about correctness in the face of concurrent updates is not a trivial task. For example, the correctness proof for a linked list implemented using elastic transactions takes up 2.5 pages in [3].

with the transaction's snapshot but does not carry any promise that the value is valid when the transaction commits. A light read tells the STM that a read value need not be valid at commit time, allowing it to return an older version.

Performance. The use of multiversions avoids false aborts and results in view transactions outperforming early release by 2× and a TL2-like STM by 7× on a linked list benchmark (which is prone to false aborts) on an Oracle T2+ processor. Moreover, the light read primitive imposes almost zero overhead on top of the version checks done at read, whereas both previous approaches need to perform some (even if minimal) bookkeeping. This is significant because in transactions that benefit from relaxed consistency checks, most read values are not critical and can be read using light reads. Therefore, (1) view transactions outperform elastic transaction by 1.13× on the linked list benchmark (despite identical abort rates), and (2) on a red-black tree workload (with no false aborts) view transactions manage to outperform early release by 1.2× and elastic transactions by 1.3×.

View pointers. How can a programmer specify the critical view of a transaction? It is possible to have a location that was originally accessed using a light read be validated at commit time by rereading it using a normal STM read. We propose simplifying this task using *view pointers*, a layer above the STM interface. View pointers are STM-aware objects the register themselves with the STM when created, and unregister when destroyed. When a view pointer is dereferenced, it uses the `light read` primitive to access the memory it points to. Whenever a transaction makes an externally visible action (like a write or commit) the locations that are currently pointed to by registered view pointers are added to the read set. While not guaranteed to work in general, using view pointers instead of standard pointers seems to work well on linked lists, search trees and similar data structures. It is interesting to characterize the conditions under which view pointers guarantee program correctness.

References

[1] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13:185–221, June 1981.

[2] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, volume 4167 of *LNCS*, pages 194–208. Springer-Verlag, Oct 2006.

[3] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. In *Proceedings of the 23rd International Symposium on Distributed Computing (DISC'09)*, volume 5805 of *LNCS*, pages 93–107. Springer-Verlag, Sep 2009.

[4] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–184, New York, NY, USA, 2008. ACM.

[5] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[6] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[7] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[8] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.