Brief Announcement: View Transactions: Transactional Model with Relaxed Consistency Checks

Yehuda Afek, Adam Morrison and Moran Tzafrir
School of Computer Science
Tel Aviv University
afek@tau.ac.il, adamx@tau.ac.il, moran.tzafrir@cs.tau.ac.il

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. As a result, view transactions outperform the prior approaches by $1.13\times$ to $2\times$ on various benchmarks.

Categories and Subject Descriptors

D.1.3 [**Software**]: Concurrent Programming

General Terms

Algorithms, Performance

1. INTRODUCTION

The transactional memory (TM) [6] paradigm promises to enable fine-grained parallelism together with ease of programming. In the TM model programmers write sequential transactions, and the TM system executes the transactions safely in parallel, *isolated* from each other, with the final result being as if the transactions had executed in some sequential one-at-a-time order (a condition known as *serializability* [8]). As a result, programmers are relieved from reasoning about all possible interleavings between fine-grained thread operations, which is notoriously difficult and error-prone [7].

Most TMs guarantee that a transaction always observes a consistent state of memory while running, even if it is doomed to abort (this is known as *opacity* [4]). Opacity is a crucial part of TM usability, since without it correct (transactified) sequential code could behave arbitrarily upon seeing inconsistent state.

The fly in the ointment is that TM implementations constantly perform *consistency checks* to guarantee serializability and opacity. These checks result in a considerable performance hit. In addition to the overhead incurred from maintaining and validating a read set (as most modern STMs do), the demand for serializability at the low level of reads and writes can lead to *false aborts* — aborts of transactions that would not have violated program semantics, had they committed. 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.

Motivated by these performance issues, researchers have proposed relaxing STM consistency checks in different ways to improve performance. *Early release* [5] augments the TM interface with a `release` primitive that lets programmers *release* previously opened objects from future checks. More recently, a novel relaxed model has been proposed in the form of *elastic transactions* [3]. An elastic transaction is a sequence of mini-transactions, each of which operates on a consistent state, but these states may not be mutually consistent. When an elastic transaction reads a value that is not consistent with its current snapshot, it may *cut* itself by committing the current mini-transaction and continuing to execute in a new mini-transaction, for which the read value is consistent.

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. Yet in these proposals a transaction with relaxed consistency checks no longer works on a snapshot. As a result, reasoning about correctness with relaxed checks becomes similar to reasoning about concurrency: programmers need to consider the inconsistent states a transaction might observe and handle them.

Contributions. We introduce *view transactions*, a new model for relaxed consistency checks. A view transaction *always operates on a consistent state (snapshot) of memory*, but may commit in a different snapshot than the one it worked on (hence the consistency relaxation). 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 runtime snapshot that needs to be validated at commit-time. View transactions improve on previous relaxed consistency approaches on two fronts, *performance* and *usability.*

9[1]The *set* abstract data type maintains a set of items and supports the operations `contains()`, `insert()` and `remove()` on the set.

2. VIEW TRANSACTIONS

It is seemingly easy to obtain a relaxed transactional model with opacity by adapting early release to a modern STM that uses per-location metadata and a global version clock (e.g., [2]). In contrast to DSTM [5], which relied on continuously validating the read set to maintain a snapshot (and therefore could violate opacity after early release removed locations from the read set), modern STMs rely only on the global clock. 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.

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 from dealing with inconsistent data, as happens in previous approaches.

To exploit multiversions, view transaction use a new *light read* primitive, which returns a value consistent 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 *ahead of time*, allowing it to return an older version. In contrast, an interface like early release communicates to the STM that a read may not need to be valid at commit time only *after the fact*, forcing the STM to abort a transaction if it tries to read an inconsistent value (since it does not know if the location will be released in the future).

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.

3. BENEFIT OF VIEW TRANSACTIONS

View transactions combine the following properties, that are not all present together in the previous relaxed transactional consistency models:

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). 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×.

Simple reasoning. Due to committed transactions working on a snapshot, reasoning about them is closer to traditional *sequential* reasoning. Additionally, view transactions are flexible enough to implement both the early release and the elastic transactional model (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 there are a subset of all possible original executions.

Opacity. 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. But to facilitate wider use of relaxed consistency checks, programmers should be relieved from worrying about these inconsistencies.

Composability. The ability to *compose* view transactions when their critical views are correctly identified by the programmer. For example, a list `contains()` that fails can place the nodes where its traversal stopped into the read set, ensuring that subsequent insertion of the item will be detected. While this requires care on the part of the programmer, it is at least *possible* while still maintaining high performance. In elastic transactions composability is not always possible, and in early release performance suffers. Composability allowed us to adapt the STAMP benchmark `vacation` to view transactions, and we found that they outperformed the STM version by $10\% - 15\%$ and early release by $5\% - 6\%$.

4. 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.