On Safety of Replicated Transactional Memory

Tadeusz Kobus, Maciej Kokociński, Paweł T. Wojciechowski

Institute of Computing Science,
Poznań University of Technology
60-965 Poznań, Poland
{Tadeusz.Kobus,Maciej.Kokocinski,Pawel.T.Wojciechowski}@cs.put.edu.pl

Abstract

Transaction Memory (TM) is a concurrency control abstraction that allows the programmer to specify blocks of code to be executed atomically. In this paper, we consider a distributed variant of TM in which transactional memory is consistently replicated on network nodes for greater availability and fault-tolerance. We argue that opacity, a standard TM safety property, is misused when applied to replicated transactional systems. In this paper we also sketch the requirements for a new safety property that can work well with all kinds of transactional systems, including replicated TM.

1 Introduction

The discussion about the desired safety guarantees of transactional memory (TM) systems is as old as the field of TM itself. Ideally, all transactions in a TM system should appear as if they were executed sequentially in a way that respects the order in which all non-overlapping transactions were originally performed. This requirement is captured by *opacity*, a well established property originally described in [5] and later explored in [6]. The system model in which opacity was defined, has often been used as a framework for new properties, proving lower bounds and impossibility results. An important result of [1] shows that opacity is necessary and sufficient to provide an illusion to the programmer, that all transactions (regardless whether they end with commit or abort) were executed sequentially.

Opacity also became the desired safety property of replicated TM systems [9] [10] [3] which emerged after TM was adopted to the distributed environment. However, as we show in this paper, opacity, in its original definition, is not general enough to be used as a correctness criterion for replicated TM. In the light of these findings, it comes as no surprise, that no formal proof of opacity has ever been published for any replicated TM. It turns out that the system model of opacity is inherently incompatible with the realms of distributed environment. In the paper we sketch the requirements for a new correctness criterion that aims at all kinds of transactional systems, including replicated TM.

2 Problems with Opacity in Replicated TM

The system model of Guerraoui and Kapalka [6] assumes that the processes interact with the transactional shared objects (called *t-objects*) only through the interface of TM. Each transaction is bound to only one process and this process cannot execute two transactions concurrently. Moreover, the transaction's code is not given as a function or code. Instead, a transaction is

executed interactively, so that the process observes the results of all transactional operations it executes.

The choice of such a system model has numerous consequences. For instance, it has been proven that it is impossible for a TM system that adheres to this model to guarantee both opacity and local progress [2]. By allowing the processes to *help* one another with execution of transactional operations, the impossibility no longer holds [12] (see also below).

The model described above is especially troublesome for replicated systems. Consider a replicated state machine (SMR) as in [11] where each replicated process executes every request on its local memory; thus, we can say that processes' state and requests' execution are *replicated*. Intuitively, if we consider the SMR requests as transactions, SMR should provide opacity: all requests are executed deterministically and sequentially, in the order they were issued by the clients. However, it is impossible to frame such an execution in the system model of opacity. There is no single process that executes a request. Instead it is executed by all processes independently on replicated t-objects. Moreover, request processing is not interactive. Each process receives the code of the request and only the return value is provided back to the client. It means that the client cannot observe the intermediate results of execution.

One could argue that in case of SMR it is easy to consider each process independently and show that all processes are exact replicas. So if execution on one process satisfies opacity then the whole system should also guarantee it. Unfortunately, it turns out that this is not true for more sophisticated replication schemes which mix different request execution modes and allow various degrees of replication (such as in partial replication). Moreover, this approach does not solve the problem of a request's code which has to be known a priori.

Consider a yet another example which features a distributed system providing a shared memory abstraction that is transparent to the programmer. One could easily construct a distributed transactional memory system that satisfies opacity and is based on this abstraction. However, potentially this system would exhibit extremely poor performance. It is clear that in order to execute transactions efficiently, we have to know how shared objects are replicated. For instance, this knowledge enables to batch in a single message several operations to be executed on the same remote shared object replica. The key to understand the problem is therefore the fact that in the distributed environment we have to explicitly care about replication. However, the original definition of opacity is incompatible with this new requirement. When we use opacity as the correctness criterion of distributed TM systems it is impossible to reason about the underlying replication scheme. It is because the definition of opacity involves only histories of interaction between the processes and the TM interface which consist of operations' invocation and response events.

On the other hand, abstracting away from the internals of TM implementation enables using opacity for all different implementations of TM, regardless whether they execute transactions optimistically or pessimistically, use backlog or shadow copies of shared objects, etc. One would expect that opacity should also be equally applicable to both local and replicated TM, as replication may be considered only an implementation detail. But unfortunately it is not. In particular, opacity does not allow processes to know transaction's code a priori, which is necessary to efficiently implement replication.

3 Looking at the TM Alternatives: Universal Construction

The concept of *Universal constructions (UC)* [7] is somewhat related to TM in the sense that both approaches aim at simplifying concurrent programming. UC, however, are primarily designed as a framework for implementing concurrent data structures. Formally, a universal construction is defined as an algorithm that produces a concurrent implementation for a shared object whose sequential specification was given as the input. The atomic execution units for

UC, which correspond to transactions in TM, are called *operations*. Contrary to transactions, operations in UC always succeed. In reality, UC may execute an operation multiple times (also by different processes), but the operation retries are transparent to the programmer. Therefore, UC acts as a contention manager featured in many TM systems. The important difference between UC and TM is that in UC the programmer cannot observe the intermediate results of operation execution and the possible operation retries.

Linearizability [8] is the correctness criterion for UC. It enforces that the concurrently submitted operations are executed as if they happened in a single point during their lifetime. More formally, for an execution history to be linearizable there has to exist an equivalent legal sequential history that respects the real-time order of operation invocations and responses. Legal histories are limited to executions in which operations on the same shared object follow its sequential specification. It means that linearizability can be perceived as a *high-level* property– only the semantics of operations (as defined by the sequential specification) is considered.

Recently linearizability was used in the context of TM. Crain et al. [4] argue that transaction retries should be transparent to the programmer. They demonstrate a software transactional memory (STM) system that guarantees exactly-once semantics for transaction execution. Their STM system is in fact a UC, where each transaction is run as a separate operation that always succeeds. In reality, each operation can be executed multiple (but bounded) number of times. Since an operation can also access local variables (beside t-objects), in case of a retry, the local variables used inside transactions are reset to their initial state. This is necessary to ensure linearizability as no information can leak from the aborted transaction runs.

This approach seems very promising in the context of replicated TM systems. However, it has some drawbacks, also for local TM. Firstly, linearizability does not guarantee consistency for live (or aborted) transactions. It means that live transactions may fall into inconsistencies and, e.g., never even try to commit. To overcome this issue, Crain et al. opted for a strong progress condition, which guarantees that every transaction commits after a bounded number of runs. However, such a progress guarantee may be deemed too strong for TM. In fact, we are not aware of any other TM system that satisfies such a stringent requirement. Secondly, hiding transaction retries from the programmer requires additional maintenance of local variable accessed by a transaction, as in case of the STM system discussed above. This additional work may result in a noticeable performance penalty. Finally, there is no common approach to proving linearizability for TM systems, as linearizability is not typically used in the context of TM. In this respect, using opacity is much easier–a typical way of proving opacity is presented in [6].

4 Conclusions and Future Work

We see that opacity is not a suitable correctness criterion for replicated TM systems. As we discussed above, linearizability is a possible replacement. However, there are several problems with this approach, which have to be addressed.

Alternatively, we intend to design a new safety property that inherits the best features of both linearizability and opacity. Similarly to linearizability, the new property would require the code of a transaction to be known a priori. However, a transaction would be able to end with abort, as in opacity. Likewise, the definition would not be limited to the high-level semantics of transactions, but it would involve explicit operations on shared (transactional) objects.

Right now we are not sure which approach of the two mentioned above is best since we only present preliminary results in this paper. Before going further we think that we (as the TM community) first have to better understand the differences between the worlds of universal constructions and transactional memory. It seems that the literature lacks a clear comparison between these two approaches in the context of used system model, offered guarantees, impossibility results, time and resource complexity, etc.

Our goal is to investigate these issues further. So far, we have developed a replicated TM system called Paxos STM [13] [9]. Our system uses novel TM algorithms that we believe are opaque. However, we came into problems trying to formally prove the correctness of our algorithms. The standard proof techniques for opacity deemed impractical in the system model that we have.

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

References

[1] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. A programming language perspective on transactional memory consistency. In *PODC*, pages 309–318, 2013.

[2] V. Bushkov, R. Guerraoui, and M. Kapałka. On the liveness of transactional memory. In *Proc. of PODC '12*, pages 9–18, 2012.

[3] N. Carvalho, P. Romano, and L. Rodrigues. Asynchronous lease-based replication of software transactional memory. In *Proc. of Middleware '10*, 2010.

[4] T. Crain, D. Imbs, and M. Raynal. Towards a universal construction for transaction-based multiprocess programs. *Theor. Comput. Sci.*, 496:154–169, July 2013.

[5] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proc. of PPoPP '08*, Feb. 2008.

[6] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[7] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proc. of PPoPP '90*, pages 197–206, 1990.

[8] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), 1990.

[9] T. Kobus, M. Kokociński, and P. T. Wojciechowski. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *Proc. of ICDCS '13*, July 2013.

[10] R. Palmieri, F. Quaglia, and P. Romano. AGGRO: Boosting STM Replication via Aggressively Optimistic Transaction Processing. In *Proc. of NCA 2010*, pages 20–27, 2010.

[11] F. B. Schneider. *Replication management using the state-machine approach*, pages 169–197. ACM Press/Addison-Wesley, 1993.

[12] J.-T. Wamhoff and C. Fetzer. The Universal Transactional Memory Construction. In *TRANSACT 11*, June 2011.

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