Generalizing the Correctness of Transactional Memory

Rachid Guerraoui Thomas A. Henzinger Michał Kapałka Vasu Singh

EPFL, Switzerland

1 Introduction

Transactional memory (TM) is a promising paradigm for concurrent programming in the multi-core era. It provides programmers a simple and familiar tool—transactions—and gives opportunities to execute those transactions concurrently in a scalable way. Ideally, programs could execute all operations on shared data within transactions, and optionally some operations on thread-local data non-transactionally. In practice, however, not all operations on shared data can be wrapped in transactions because of either performance concerns or requirements of interoperability with legacy software components. It is thus not surprising to see a large body of research dedicated to exploring the various models of interactions between transactions and non-transactional code, and implementing those models, e.g., [6, 2, 7, 1].

The mutual interaction between transactions is formalized by the notion of *opacity* [3], which is indeed ensured by most TM implementations. Opacity requires that transactions should "look like" they executed in some sequential order, consistent with their real-time order, and no transaction (even an aborted one) should ever observe an inconsistent state of the system. At a high-level, this gives programmers an illusion that there is no concurrency between transactions, which resembles the intuitive model of critical sections protected by a single global lock.

The interactions between non-transactional operations are expressed using a *memory model*. While a TM can be implemented in a way to ensure opacity for transactions, there is little one can do (on a given platform or run-time environment) to change the underlying memory model. Hence, it is desirable to define opacity *parametrized* by a memory model. This would allow for exploring the influence a memory model has on a TM system, e.g., by proving inherent impossibilities and complexity lower bounds that hold only for some memory models, or showing that certain TM implementation techniques are fundamentally expensive, or even impossible, to use regardless of the memory model.

Thread 1	Thread 2
begin-transaction	
$x = 1$	
$y = 1$	$r_1 = x$
commit	$r_2 = y$

Figure 1: Can $r_1 = 1$ and $r_2 = 0$? It depends on the memory model (initially $x = y = 0$).

There are many ways in which one can specify the semantics of possible interactions of transactions and non-transactional code, e.g., by giving sample executions [9], mapping transactions to some existing language constructs (usually critical sections) [7], adding rules to a concrete memory model (usually the Java one) [2], or devising a set of rules that should, in principle, hold irrespective of the memory model[1] [8]. The discussion can be simplified if one takes a programmer-centric approach: assumes a very strong memory model and a high level of isolation of transactions from non-transactional operations, but only for programs that follow a certain set of rules (i.e., programs that are free of race conditions) [1]. Each of those approaches is valid, but each of them either limits the scope to a single environment (or memory model) or leaves too many aspects unspecified or imprecise.

2 Our Goal

This paper is a first step in a quest for a general formal framework for describing the interactions between transactions and non-transactional code. Our approach is to assume opacity as a correctness condition for transactions, and to parametrize it

[1] Which is not entirely trivial—e.g., the "no out-of-thin-air values" rule given in [8] would be too strong for many memory models, especially the hardware ones.

with respect to a memory model. The task is non-trivial because (1) we need a definition of a memory model that is very precise but still universally applicable (e.g., it can encompass hardware as well as higher-level memory models), and (2) we need a formal way to specify the interactions between transactions and non-transactional code without making assumptions about the memory model. We would also like a property that is strong (from a user's perspective) but otherwise as universally applicable as possible. For example, parametrized opacity (like opacity) should be implementation-agnostic and should allow for transactional objects with semantics richer than that of simple read-write variables (which could help describing, e.g., TMs that implement transactional boosting [4] or similar techniques).

Our quest is guided by the following requirements:

1. **Opacity for purely transactional code.** Whatever the memory model is, executions that are purely transactional must ensure opacity. Indeed, the semantics of transactions should be intuitive and strong—in the end, we want TMs to be as easy to use as coarse-grained locking.

2. **Efficiency of purely non-transactional code.** Executions that are purely non-transactional have to adhere to the given memory model. In particular, parametrized opacity should not strengthen the semantics of non-transactional operations. The motivation here is to avoid a framework that would inherently require non-transactional operations to be instrumented with additional memory fences or software barriers, even for very weak memory models.

3. **Isolation of transactional code from non-transactional code.** Transactions should appear, both to other transactions and non-transactional operations, as if they were executed *instantaneously*. In particular, isolation of transactions should be respected, regardless of the memory model. That is, first, the intermediate computations of transactions, or updates by aborted transactions, should never be visible to non-transactional operations, and, second, the non-transactional operations concurrent to a transaction should appear as if they happened before or after this transaction (e.g., if a transaction reads a variable x twice in a row, it should read the same value from x both times, even in presence of concurrent updates to x).

The goals 2 and 3 may seem to be contradictory: on the one hand, we want to avoid instrumenting non-transactional code, but, on the other hand, we give a requirement that resembles strong isolation[2] [6, 1]. Consider, for example, the execution depicted in Figure 1 (adapted from [2]). The transaction executed by thread 1 updates variables x and y. If the TM is implemented in software, those two updates are not executed at the same time:[3] there is a window in which only one of the variables is updated. Hence, if the non-transactional code executed by thread 2 is not instrumented with special barriers, thread 2 could read $r_1 = 1$ and $r_2 = 0$, which could seem to be against requirement 3. Note, however, that this really depends on the memory model. Indeed, if the memory model allows independent reads to be reordered, then the read of y executed by thread 2 may be ordered before the read of x. It is then clear that the read of y may return 0 (as if it was executed before the transaction of thread 1 started) and the read of x may return 1 (as if it was executed after the transaction committed).

A memory model (transaction-aware or not) should be intuitive to the programmer and, at the same time, should support common compiler optimizations. For example, the Java memory model (JMM) provides a simple definition of data race freedom and guarantees strong (and intuitive) semantics for programs that are indeed data race free. Yet, it supports common compiler optimizations like swapping loads or stores, or eliminating irrelevant loads.

Open questions. The example in Figure 1 illustrates also one of the interesting questions that we may be able to answer once we have a proper theoretical framework: what are the memory models for which one can implement the strong guarantee of parametrized opacity without incurring any overhead on non-transactional code? More generally, given any memory model M, what are the lower bounds on the time and space complexities of a TM implementation that ensures opacity parametrized by M? Can this implementation be obstruction-free or even lock-free, or does it have to make transactions wait for each other (like in the

[2] Originally called "strong atomicity".

[3] Assuming an update-in-place TM. Indirection-based TMs would require modifications to non-transactional code in any case.

solution for privatization-safety presented in [9])? Does the non-transactional code have to be instrumented, and, if yes, what is the inherent cost of this instrumentation?

3 Parametrized Opacity

We describe here a first attempt at defining parametrized opacity. Due to space constraints, some (less important) details are omitted.

We consider a shared-memory system of n processes (threads) $p_1, \ldots, p_n$ that communicate by executing operations (e.g., read, write, compare-and-swap) on (shared) objects. Furthermore, each process p_i can issue the following special operations: start a new transaction, and commit or abort the transaction currently active at p_i.

Histories. A *history* is a sequence of operations (together with their return values) executed in a given run. Given a history H, we denote by $H|p_i$ and $H|x$ the longest subsequences of all operations in H executed, respectively, by process p_i or on object x.

Given a history H, we define the precedence relation $\prec_H$ (the real-time order) of the operations in H, such that $o \prec_H o'$ if:

- o and o' are executed in some transactions T and T', respectively, T is committed or aborted in H, and the last operation of T precedes the start operation of T' (i.e., T is executed, and completed, entirely before T' starts), or
- o is executed in a transaction T, o' is a non-transactional operation, both o and o' are executed by the same process, and o' follows the last operation of T in H, or
- o is a non-transactional operation, o' is executed in a transaction T', both o and o' are executed by the same process, and o precedes the start operation of T', or
- there exist a sequence $o_1, \ldots, o_m$ of operations in H such that $o \prec_H o_1 \prec_H o_2 \ldots \prec_H o_m \prec_H o'$ (transitive closure).

We say that histories H and H' are *equivalent* if they contain (a) the same transactions, executing the same operations, and (b) the same non-transactional operations.

We say that a history S is *sequential* if, for every transaction T in S, every operation between the start operation of T and the last operation of T is also an operation of T. We say that a sequential history S *respects a partial order* $<$ if transactions and non-transactional operations in S follow order $<$.

Memory model. A *memory model* is a tuple $M = (\tau, R)$, where τ is a function that maps any operation to a sequence of (zero or more) operations, and R is a function that maps any sequence of operations and a process identifier into a partial order relation on those operations. Intuitively, τ maps an operation to its internal representation (e.g., in hardware or a run-time environment). For instance, a write to a 64-bit memory word might be, on some systems, executed as two writes to its 32-bit parts. In the extreme case when a memory model does not give any guarantees for unsynchronized code (e.g., in languages like C/C++), τ can map every write to a special *havoc* operation followed by the write (a read that follows a havoc operation can return any value). The function R expresses the ordering constraints of operations, as seen by every process in the system.

If $M = (\tau, R)$ is a memory model and H is a history, then $\tau(H)$ denotes the history obtained from H by replacing every non-transactional operation o in H with sequence $\tau(o)$.

Semantics of shared objects. We use the concept of a *sequential specification* to describe the semantics of objects, as in [10, 5]. Intuitively, the sequential specification of an object x is the set of all sequences of operations on x (together with their return values) that could be generated by a single process accessing x non-transactionally without any concurrency. For example, the sequential specification of a read-write variable is the set of all sequences or reads and writes in which every read returns the value written by the latest preceding write.[4]

We say that a sequential history S is *legal*, if, for every object x, the history $S|x$ is in the sequential specification of x.

Let S be any sequential history and o be any operation in S. We denote by $visible(S, o)$ the longest subsequence of S in which every transaction T is committed, except possibly the transaction that executes o. (Intuitively, updates of operations of a non-committed transaction T can be visible only

[4]The notion of a sequential specification was originally used to describe only atomic objects. We do not assume that all objects are atomic.

to other operations of T.) We say that an operation o in S is *legal* in S if history $visible(S', o)$ is legal, where S' is the prefix of S that ends with operation o.

The definition of parametrized opacity. We say that a history H ensures *opacity parametrized by a memory model* $M = (\tau, R)$, if there exists a total order $\ll$ on the set of transactions in H, such that, for every process p_i, there exists a sequential history S_i equivalent to history $\tau(H)$, such that (a) S_i respects partial order $\ll \cup \prec_H \cup R(H, p_i)$, and (b) every operation in $S_i|p_i$ is legal in S_i.

References

[1] Luke Dalessandro and Michael Scott. Strong isolation is a weak idea. In *TRANSACT*, 2009.

[2] Dan Grossman, Jeremy Manson, and William Pugh. What do high-level memory models mean for transactions? In *Memory System Performance and Correctness*, 2006.

[3] Rachid Guerraoui and Michał Kapałka. On the correctness of transactional memory. In *PPoPP*, 2008.

[4] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*, 2008.

[5] Maurice Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), 1990.

[6] Milo M. K. Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), 2006.

[7] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Practical weak-atomicity semantics for Java STM. In *SPAA*, 2008.

[8] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. Ordering-based semantics for software transactional memory. In *OPODIS*, 2008.

[9] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. Technical report, Computer Science Department, University of Rochester, 2007.

[10] William E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2), 1989.