

# Safer Open-Nested Transactions Through Ownership

Kunal Agrawal I-Ting Angelina Lee Jim Sukha

MIT Computer Science and Artificial Intelligence Laboratory

{kunal\_ag, angelee, sukhaj}@mit.edu

**Categories and Subject Descriptors** D.2.1 [Software Engineering]: Requirements/Specifications — Methodologies; D.3.3 [Programming Languages]: Language Constructs and Features — Concurrent programming structures

**General Terms** Design, Languages

**Keywords** Transactional Memory, Open-nested Transactions, Ownership Types, Abstract Serializability, Ownership-aware Transactions, Type System

## Introduction

Transactional memory (TM) (4) is meant to simplify concurrency control in parallel programming by providing a transactional interface for accessing memory; the programmer simply encloses the critical region inside an `atomic` block, and the TM system ensures that that section of code executes atomically. A TM system enforces atomicity by tracking the memory locations that each transaction in the system accesses, finding transaction conflicts, and aborting and possibly retrying transactions that conflict. TM guarantees that transactions are *serializable* (10), that is, transactions affect global memory as if they were executed one at a time in some order, even if in reality, several executed concurrently.

Recently, *open-nested transactions* (5; 8) have been proposed as a way to increase concurrency in transactional programs. Conceptually, when an open-nested transaction  $Y$  (nested inside transaction  $X$ ) commits,  $Y$  makes its changes directly to memory and discards its readset and writeset.<sup>1</sup> This mechanism is henceforth called the *open-nested commit mechanism*. This commit mechanism is part of the *open-nesting methodology*, where the programmer considers  $Y$ 's internal memory operations to be at a “lower level” than  $X$ ; therefore  $X$  should not care about the memory accessed by  $Y$  when checking for conflicts. Open-nesting methodology requires *abstract locks* to protect the abstract operation performed by  $Y$  and *compensating actions* to undo the actions of  $Y$  if  $X$  subsequently aborts. Moss in (7) illustrates use of open nesting with an application that uses a B-tree. In (9), Ni et. al describe a software TM system that supports the open-nesting methodology.

An unconstrained use of the open-nested commit mechanism can lead to anomalous program behavior (1) that can be tricky to reason about. Since the open-nested commit mechanism is part

<sup>1</sup>When a closed-nested transaction  $Y$  commits,  $Y$  merges its readset and writeset into  $X$ 's readset and writeset.

of the methodology, it might seem that using the open-nesting methodology is complicated. Although researchers have demonstrated specific examples that safely use an open-nested commit mechanism, the literature on TM offers relatively little in the way of formal programming guidelines which one can follow to have *provable* guarantees of safety when using open-nested commits. Moreover, since these working examples require only two levels of nesting, it is not obvious how one can correctly use open-nested commits in a program with more than two levels of abstraction.

We believe that one reason for apparent complexity of open nesting is that the mechanism and methodology make different assumptions about memory. Consider a transaction  $Y$  open-nested inside transaction  $X$ . The open-nesting methodology requires that  $X$  ignore the “lower-level” memory conflicts generated by  $Y$ , while the open-nested commit mechanism will ignore *all* the memory operations inside  $Y$ . Say  $Y$  accesses two memory locations  $\ell_1$  and  $\ell_2$ , and  $X$  does not care about changes made to  $\ell_1$ , but does care about  $\ell_2$ . The TM system can not distinguish between these two accesses, and will commit both in an open-nested manner, leading to anomalous behavior. In fact, specific uses of open nesting that researchers describe (3; 9) work because they exhibit a clean separation of the data accessed by an outer transaction and its (nested) inner transaction.

The focus of this research project is to bridge the gap between memory-level mechanisms for open nesting and the high-level view by explicitly integrating the notions of *transactional modules* (called Xmodules) and *ownership* into the TM system. We propose an alternative *ownership-aware commit mechanism*, which is a compromise between an open-nested and closed-nested commit. In addition, we propose a set of concrete guidelines for data-sharing and interactions between Xmodules. If the programmer follows these guidelines, then we prove that a TM system using ownership-aware commit mechanism guarantees “serializability-by-modules” which is a generalization of “serializability-by-levels” used in database transactions. We believe an ownership-aware TM system allows programmers to safely use a methodology similar to open nesting<sup>2</sup> because the runtime's behavior more closely reflects the programmer's intent, and because ownership imposes additional structure that allows a language and runtime to enforce properties needed to provide provable guarantees of “safety” to the programmer.

In this abstract, we informally explain Xmodules, the ownership-aware commit mechanism, and state our theoretical results. For more information, see <http://supertech.csail.mit.edu/~kunal/safeTech.pdf>

## Xmodules and Ownership

We assume that programs are organized into a set  $\mathcal{N}$  of Xmodules. Each Xmodule  $A \in \mathcal{N}$  has some number of (public) methods

<sup>2</sup>The programmer still has to specify abstract locks and compensating actions to complete the methodology

and a set of memory locations associated with it. The methods of  $A$  represent services that  $A$  can provide to other Xmodules. Each Xmodule  $A$  has an owner (which is another Xmodule) that essentially has control over which other Xmodules can use the services provided by  $A$ . The ownership relation between Xmodules forms a tree, where an Xmodule is owned by its parent in the tree, and the root of the tree is a special Xmodule called `world`.

To have well defined levels of abstraction, we impose ordering constraints among sibling Xmodules in the tree, and call the resulting ordered tree the *module tree*, denoted by  $\mathcal{D}$ . Each Xmodule is assigned a `level` according to its position in the tree as follows: visit the nodes in a left-to-right depth-first search order and assign levels in a descending order. Therefore `world` has the maximum `level`. A smaller `level` number corresponds to a lower-level module and vice versa.

We use the module tree  $\mathcal{D}$  to restrict the sharing of data between Xmodules and to limit the visibility of Xmodule methods according to the rules given below.

DEFINITION 1. For a program with a module tree  $\mathcal{D}$ ,

1. An Xmodule  $A$  can only directly access memory that it owns, or memory that an ancestor Xmodule  $B$  owns.
2. A method from  $A$  can call a method from  $B$  only if  $B$  is the child of some ancestor of  $A$ , and if  $B$ 's `level` number is smaller than  $A$ 's `level` number, i.e.,  $B$  is “to the right” of  $A$  in  $\mathcal{D}$ .

Intuitively, in the first rule of Definition 1,  $A$  might access memory from an ancestor Xmodule  $B$  because  $B$  passed in that data to the lower-level Xmodule  $A$ . Since all of  $A$ 's ancestors in  $\mathcal{D}$  have larger `level` number than  $A$ , this first rule prohibits a transaction from module  $A$  from directly accessing any “lower-level” memory. The second rule enforces that an Xmodule can only call methods of some (but not all) “lower-level” Xmodules.

Whenever an Xmodule calls a method from an Xmodule from a lower-level, the method call generates a new nested transaction.<sup>3</sup> When ownership-aware TM commits a nested transaction, like with open nesting, it commits some memory locations globally. Ownership-aware TM uses a commit mechanism that differs slightly, however, from the ordinary open-nested commit.

### Ownership-aware Commit Mechanism

The ownership-aware commit mechanism can be thought of as a combination of an open-nested commit and a closed-nested commit. When a transaction commits in the ownership-aware TM, it commits memory differently depending on the owner of the memory location. If a transaction  $T$  belonging to Xmodule  $A$  commits, it commits all the memory locations belonging to  $A$  in an “open-nested” manner; the changes are committed directly to memory, and the TM system no longer detects conflicts on these memory locations. It commits all other memory locations in the “closed-nested” manner; the memory locations are propagated to the parent transaction's readset/writeset. An ownership-aware commit exhibits arguably more natural semantics than an open-nested commit in cases where a nested transaction and its parent share data.

### Theoretical Results

We prove that TM with ownership-aware commit mechanism guarantees a correctness condition for transactions which we refer to as *serializability by modules*. Informally, this correctness condition considers one Xmodule  $A$  at a time (starting with the lowest-

<sup>3</sup> Ownership-aware TM permits an exception to Definition 1 in the case of callbacks, where an Xmodule invokes methods of an ancestor Xmodule. This case is modeled as the descendant Xmodule directly accessing memory owned by its ancestor, and these callback methods do not generate nested transaction instances.

level Xmodule), and checks that the transactions of Xmodule  $A$  are serializable if we ignore all operations which access memory owned by Xmodules which have a lower level than  $A$ . This definition is closely related to the database definition of serializability by levels. Our theoretical result provides provable guarantees for programs which conform to the rules in Definition 1 and which are executed on an ownership-aware TM system. Thus, an ownership-aware commit arguably exhibits clean semantics which are easier for programmers to reason about.

In general, a TM system with only closed or flat nesting does not enter semantic deadlock.<sup>4</sup> On the other hand, a TM system with open-nesting can enter semantic deadlock because transactions in the process of aborting may have to execute compensating actions (for nested transactions) before they finish aborting. This problem exists for ownership-aware TM as well. However, we prove that ownership-aware TM is free from semantic deadlock if a compensating action for a transaction  $Y$  (nested inside  $X$ ) does not access any memory owned by  $X$ 's Xmodule or its ancestors unless the memory location is already in  $X$ 's readset or writeset.

### Acknowledgments

We would like to thank Charles E. Leiserson, Derek Rayside and Martin Rinard of MIT CSAIL and James Noble of Victoria University of Wellington for helpful discussions. This research was supported in part by NSF Grants CNS-0615215 and CNS-0540248 and a grant from Intel Corporation.

### References

- [1] K. Agrawal, C. E. Leiserson, and J. Sukha. Memory models for open-nested transactions. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, October 2006. In conjunction ASPLOS.
- [2] B. Chandrasekhar, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, January 2003.
- [3] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional collection classes. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 56–67, New York, NY, USA, 2007. ACM Press.
- [4] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 289–300, 2003.
- [5] A. McDonald, J. Chung, B. D. Carlstrom, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2006.
- [6] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, MA, USA, 1985.
- [7] J. E. B. Moss. Open nested transactions: Semantics and support. In *Proceedings of the Workshop on Memory Performance Issues (WMPI)*, Austin, Texas, Feb 2006.
- [8] J. E. B. Moss and A. L. Hosking. Nested transactional memory: Model and architecture sketches. In *Science of Computer Programming*, volume 63, pages 186–201. Elsevier, Dec 2006.
- [9] Y. Ni, V. Menon, A. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPOPP*, Mar. 2007.
- [10] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

<sup>4</sup> A semantic deadlock is different from a livelock. For a formal definition, see the full report.