

Transactions and Composability:

Transactional Memory Considered Harmful?

Suresh Jagannathan

Jan Vitek

The prospect of a miracle cure to the multi-core challenge has caused a freeing frenzy in academia and industry. At the center of the melee is the concept of *Transactional Memory*. TM abstractions permit logically concurrent access to shared data, but ensure through some combination of hardware, compiler, and runtime support that such accesses do not violate intended serializability invariants. By doing so, pernicious errors such as data races can be eliminated, and, if one is to believe the advance publicity, with performance improvements. Currently, no implementation has been able to provide performance improvement on realistic workload, and in some cases there can be significant decrease in performance.

Atomicity and isolation properties make transactions **composable** with one another: the execution of one transaction can't affect the visible behavior of another. If transactions were equally composable with other realistic language features, it would be an important benefit that would distinguish transactions from lock-based concurrency control primitives.

Unfortunately, there are many ways to compromise composability. Typically, a transaction brackets a computation whose memory operations are logged, and checked when the transaction is about to commit. Performance benefits and correctness properties of transactions critically depend upon a detailed understanding of computation they encapsulate. Gaining this understanding jeopardizes composability, and imposes burdens on the programmer not too dissimilar to the ones they face when programming with locks.

Because computations wrapped within transactions can be long-lived, transactional execution can incur significant overheads. To determine when such execution is not likely to hinder performance requires detailed knowledge about the memory operations the computation performs. However, useful abstractions like procedures or objects make such assessment difficult. Executing a call to procedure P within a transaction may lead to high logging overheads (especially if P turns out to be long lived), and make the cost of aborting prohibitive. Understanding these tradeoffs, however, requires knowledge of P 's internal actions, thus breaking abstraction. Transactions are thus not *time-composable*.

Transactions do not compose with other concurrency control mechanisms. The visibility rules dictating updates within lock-based critical sections are different from the visibility rules that govern transactional execution. If a transaction T executed by thread t enters and exits a critical section protected by lock L , other threads that subsequently acquire L would expect to see t 's effects performed within this section. Unfortunately, these visibility rules violate the atomicity guarantees provided by T . In other words, transactions can neither seamlessly co-habit with, or replace lock-based abstractions without imposing non-trivial constraints on the actions they perform.

Transactions don't compose with abstractions that explicitly initiate communication. Signaling mechanisms, producer/consumer pipelines, and message-passing primitives all require the transfer of data from one executing thread to another. If these mechanisms were encapsulated within a transaction, such transfers would be prevented because of the isolation guarantees transactions enforce; program correctness may therefore be jeopardized. Identifying when such uses occur is complicated because they may manifest beneath many software layers; handling them is complicated because the two kinds of abstractions take fundamentally divergent views of the role of isolation in their definitions.

Transactions and multi-threading is yet another instance in which a simple characterization of transactions is insufficient to guarantee composability. If a transaction encapsulates multiple thread creation points, the success of these threads now becomes dependent upon the successful commit of the transaction. Long-lived or non-terminating threads can therefore prevent transactions from committing, and from other threads encapsulated within such transactions from completing.

Open nesting is inherently non-composable. The issue of supporting open nesting in a composable way remains an open problem.

While many of these composability issues hold for lock-based abstractions as well, their manifestation within a transactional framework is more severe because (a) *transactional abstractions have been promoted as being cleaner and simpler to use than their lock-based counterparts*; and (b) *solutions to these issues are likely to require substantial code and algorithm restructuring, complicating the deployment of transactional machinery for real-world applications*.

Addressing these difficulties is a key challenge for transactional research going forward. We suspect that there is no simple "one-shoe-fits-all" design that can effectively satisfy all these concerns. Rather, we believe that there is a need for a spectrum of transactional designs that provide successively greater degrees of composability, with correspondingly weaker atomicity and isolation guarantees.