

Language-Level Semantics should guide Transactional Memory Research

Jan-Willem Maessen, Sun Microsystems Laboratories
JanWillem.Maessen@sun.com

From the very first, transactions have been the fundamental language-level synchronization operation in Project Fortress. We have been unabashedly ambitious in the demands we make of transactional memory implementations. Our process has been very simple: define clean, simple semantics for language constructs in the presence of transactional memory; relax language-level guarantees just enough that still-unsolved problems will not prevent the construction of working implementations; find working solutions to these still-unsolved problems.

Transactions in Fortress

The chief promise of transactional memory is a compositional programming style which makes it easy to exploit as much concurrency as possible. However, it is all too easy to lose compositionality unless the semantics of transactions are carefully drawn. In particular, it should be possible for the programmer to use a library of functions without knowing or caring whether they make internal use of transactional atomicity—and to manipulate a data structure defined by another library atomically.

Ubiquitous Transactions: The language must not second-guess the programmer's need for synchronization. Any and all mutable locations may be subject to transactional access. Every function ought to be callable in an atomic context (but see semantic challenges, below; irrevocable actions must be ruled out). Transactions must be allowed to touch as much data as they like and run for as long as is necessary (though we must of course make programmers aware of the loss of concurrency and efficiency this will entail). It must be possible to use the usual language mechanisms to allocate and free memory from within a transaction.

Strong Transactions: The isolation and atomicity of transactions must extend to non-transactional uses of the same data. We already presume that most code might be run in a transactional context; the additional complexity of using transactional-style loads and stores everywhere should be small. Note, however, that efficient implementation of non-transactional reads and writes must be addressed. There are more efficient techniques than starting a transaction, doing a single operation, and then committing.

Functional Style: We want to encourage our programmers to avoid mutable state. This is not, at first blush, a statement about transactions—but when we program in a functional style, we do not need to reason about the effects of concurrency at all.

In particular, the side-effect-free portion of a language need not concern itself with transactional consistency at all, so long as we guarantee that immutable data can never be seen in a partially-initialized state.

Abortable Atomicity: The ability to abandon a running transaction and roll back its effects provides a powerful mechanism to enforce high-level software invariants. We believe in abortable atomicity as an alternative to condition variables and other synchronization idioms. By using atomicity in this way we hope to avoid problems with priority inversion, lock convoying, and the like; it finally becomes possible to make meaningful and simple assertions about thread priority. However, this imposes rather stringent conditions on the TM implementation—and it opens up the possibility of deadlock or livelock, which can be avoided in the absence of abortability.

Internal Parallelism: The functions we are attempting to run atomically may (without our knowledge) contain embedded parallelism. It should be possible to exploit this internal parallelism. We take a nested-worlds view, in which parallel threads within a transaction can synchronize with one another using nested transactions. We're still working out how this might be implemented efficiently (it is not difficult to come up with a working but inefficient implementation). In particular, it seems to be difficult to provide fairness guarantees to nested parallel threads, or to allow them to interact with certain kinds of abortable synchronization.

Nested Transactions: The challenges of transactional nesting are consequences of the other decisions we have taken. If a nested transaction aborts in software, it should only cause the innermost transaction to fail. Moreover, if we permit nested parallelism each nested transaction must have an independent existence; we can no longer flatten transactions and expect consistent-but-inefficient behavior.

Exception Behavior: Having accepted abortable atomicity as a desirable mechanism for enforcing program consistency, uncaught exceptions must abort a transaction. Because we encourage a functional style from the outset, it is perfectly reasonable to imagine immutable state being preserved in an exception object. This combines the power of abortable atomicity with the ability to diagnose and address causes of failure. It is our feeling that disagreements on transactional exception semantics actually stem from more fundamental philosophical differences on the appropriateness of exposing abortable atomicity at the language level.

Challenges

Revocable Actions and Open Nesting: Often we wish to take an action which touches memory or calls out to the operating system, but does so in a semantically transparent or revocable way. Open nesting has been proposed as a mechanism for doing so. In our view, the real challenge is defining what we mean for an action to be “semantically transparent.” A classic example use of open nesting is to generate a unique identifier by bumping a counter. However, a program which exploits properties of the integer returned *other* than its uniqueness can run into trouble. Are there language-level mechanisms we can use to guarantee that certain actions really *must* be semantically transparent from the perspective of the programmer? What is the right way to expose these mechanisms at the language level?

Irrevocable Actions: So long as we have dynamic transactions which admit the possibility of failure, we simply cannot permit irrevocable actions to be taken during a transaction. It is easy for the system to declare that any given action is irrevocable, and to build a type system which prevents irrevocable actions during a transaction. However, there may be fewer truly irrevocable actions than we realized. For example, if we permit open-ended buffering it is possible to perform I/O within a transaction. However, we then need to worry about the *interactions* of these actions. For example, we would like to forbid writing a request to a pipe and then attempting to read a response within the same transaction.

Hand-over-hand Synchronization: In the past we have used fine-grained hand-over-hand locking to avoid bottlenecks while traversing lists and trees. We can use *early release* of transactional reads to the same effect. However, this means that there is no longer a simple serialization-based semantics for transactions; it is worth trying to understand what semantics we might reasonably expect instead. We advocate a semantics in which a later access to a location subject to early release revokes the release, preserving isolation at the cost of concurrency.

Implementation notes

Read and write sets are a powerful abstraction for reasoning about the feasibility of our semantics. If we can describe the operational behavior of our desired semantics in terms of transactional read and write set manipulations, we can implement those semantics. The challenge, therefore, is to choose underlying representations (and design appropriate hardware support) which will do so efficiently.

We are pragmatic about the use of blocking in TM implementations. Individual transactional operations must be made as cheap as possible—especially given that we imagine using transactional memory access everywhere! The best way to avoid the costs of blocking and concurrency management is to make transactions run faster and thus overlap less.

Garbage collection simplifies the life of the concurrent programmer. The greatest complexity in concurrent algorithms is devoted to addressing the ABA problem. Garbage collection solves the ABA problem for pointers once, in a modular and transparent way. It imposes a programming discipline, but one which is simpler and far less onerous than in a system with explicit memory management.

Many of the techniques which are used in garbage collection—particularly the notion of read and write barriers, and of safe points—appear in a slightly different form in TM implementation. It should be possible to integrate the two mechanisms so that overheads are shared rather than additive. There is a shared design space which has not been well explored; for example, does TM make read-barrier-based GC techniques more attractive? Can we use GC barrier tricks to make the common case of conflict-free TM faster? What compiler optimizations carry across?

By making use of a managed run time, we can further simplify the implementation of transactional memory. For example, embedding transactional memory in C or C++ requires frequent validation of transactions in progress in order to avoid segmentation faults and infinite loops. If the run time is able to detect and recover from these inconsistencies we can shift work to those transactions which have already failed and away from ones which will succeed.

Finally, we are worried about the million-thread universe, where the design tradeoffs are very different from those being explored today. We are especially leery of TM implementations whose efficiency relies on maintaining small amounts of heavily-shared state (such as global counters), or which require state which grows linearly in the number of threads.