Language-level Transactions for Modular Reliable Systems

C. Scott Ananian Martin Rinard
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
{cananian,rinard}@csail.mit.edu

The transaction model is a natural means to express atomicity, fault-tolerance, synchronization, and exception handling in reliable programs. A (lightweight, in-memory) transaction can be thought of as a sequence of program loads and stores which either ***commits*** or ***aborts***. If a transaction commits, then all of the loads and stores appear to have run atomically with respect to other transactions. That is, the transaction's operations appear not to have been interleaved with those of other transactions or non-transactional code. If a transaction aborts, then none of its stores take effect and the transaction can be safely restarted, typically using a backoff algorithm to preclude live-lock. A subset of the traditional ACID database semantics are provided.

Although transactions can be implemented using mutual exclusion (locks), we present algorithms utilizing non-blocking synchronization to exploit optimistic concurrency among transactions and provide fault-tolerance. A process which fails while holding a lock within a critical region can prevent all other non-failing processes from ever making progress. It is in general not possible to restore the locked data structures to a consistent state after such a failure. Non-blocking synchronization offers a graceful solution to this problem, as non-progress or failure of any one thread or module will not affect the progress or consistency of other threads or the system.

Implementing transactions using non-blocking synchronization offers performance benefits as well. Even in a failure-free system, page faults, cache misses, context switches, I/O, and other unpredictable events may result in delays to the entire system when mutual exclusion is used to guarantee the atomicity of operation sequences; non-blocking synchronization allows undelayed processes or processors to continue to make progress. Similarly, in real-time systems, the use of non-blocking synchronization can prevent *priority inversion* in the system by allowing high priority threads to abort lower priority threads at any point.

We show how to integrate non-blocking transactions into an object-oriented language, "transactifying" existing code to fix existing concurrency bugs and using transactions for modular fault-tolerance, backtracking, exception-handling, and concurrency control in new programs.

We propose the use of compiler-supported "atomic" blocks to specify synchronization. This is less error-prone than manual maintenance of a locking discipline: deadlocks may be introduced when locks are not acquired and released in a highly disciplined manner, and the specification of locking discipline cuts across module boundaries. Races are common when multiple shared objects are involved in an operation, each with its own lock. We provide several examples of such problematic locking code. A non-blocking transaction implementation prevents inadvertent deadlocks, and `atomic` declarations implemented with the transaction mechanism can extend across method invocations and module boundaries to protect multiple objects involved in an operation without allowing races between them. An optimistic non-blocking implementation provides performance improvements over locking strategies in some cases as well.

Language-level transactions are used as a general exception-handling and backtracking mechanism. Instead of forcing the programmer to manually track changes made to program state in order to implement proper fault recovery, we can handle the exception using transaction rollback to automatically restore a safe program state, even if the fault occurred in the middle of mutating shared objects. An efficient and graceful transaction mechanism integrated into the programming language encourages a robust programming style where recovery and retry after an unexpected condition is made simple and faults and recovery do not break abstraction boundaries.

We describe an efficient pure-software transaction mechanism we have implemented for programs written in Java. We also discuss our design and simulation (with Asanović, Kuszmaul, Leiserson, and Lie) of minimally-intrusive architecture extensions which allow most transactions to complete with near-zero overhead. Unlike previous hardware approaches, our scheme is scalable and supports transactions of unlimited size, although performance is best for transactions which fit in local cache. Finally, we describe our hybrid hardware-software scheme combining the speed hardware provides for small transactions with the flexibility the software implementation allows for large or long-lived transactions.

Integrating transactions into the programming language and implementing them with the high-efficiency techniques described enables the creation of software with higher reliability. Synchronization is more robust and its specification is modular and less error-prone, and faults and exceptions in general can be soundly handled with low overhead using the transaction mechanism.

This research was supported by DARPA/AFRL Contract F33615-00-C-1692.