DISC '07

Transaction Safe Nonblocking Data Structures *

Virendra J. Marathe, Michael F. Spear, and Michael L. Scott

Department of Computer Science, University of Rochester
Rochester, NY 14627-0226 USA
{vmarathe, spear, scott}@cs.rochester.edu

This brief announcement focuses on interoperability of software transactions with ad hoc nonblocking algorithms. Specifically, we modify arbitrary nonblocking operations so that (1) they can be used both inside and outside transactions, (2) external uses serialize with transactions, and (3) internal uses succeed if and only if the surrounding transaction commits. Interoperability enables seemless integration with legacy code, atomic composition of nonblocking operations, and the equivalent of hand-optimized, closed nested transactions.

The key to transaction safety is to ensure that memory accesses of operations called from inside a transaction occur (or appear to occur) if, only if, and when the surrounding transaction commits. We do this by making writes manifestly speculative, with their fate tied to that of the transaction, and by logging reads for re-validation immediately before the transaction commits. (Because correct nonblocking code is designed to tolerate races, additional, intermediate validation is not required.) When called from outside a transaction, operations behave as they did in the original nonblocking code, except that they aggressively abort any transaction that stands in their way. Operations inside a transaction similarly abort transactional peers. They are unaware of nontransactional peers.

We provide nonblocking objects with "transaction aware" versions of references and other basic primitive types such as integer, long, etc. These provide Get, Set, and CAS operations, which the programmer uses instead of conventional accesses. If called inside a transaction, Get logs the target location for later validation; Set and CAS speculatively modify the target location. Changes become permanent at transaction commit time. If called outside a transaction, all three operations "clean up" any encountered speculative updates, aborting conflicting transactions if necessary. Given correct nonblocking code, the changes required to create a transaction-safe version are mechanical.

To make a type transaction-aware, we must be able to distinguish between real and speculative values. For some types (e.g., pointers in C) we may be able to claim an otherwise unused bit, or use features such as *runtime type identification* in strongly typed languages such as Java. For others we may use a sentinel value to trigger address-based lookup in a separate metadata table. With support for transaction aware primitives, we expect that construction of transaction safe versions of nonblocking algorithms would require little or no

* This work was supported in part by NSF grants CNS-0411127 and CNS-0615139, equipment support from Sun Microsystems Laboratories, and financial support from Intel and Microsoft.

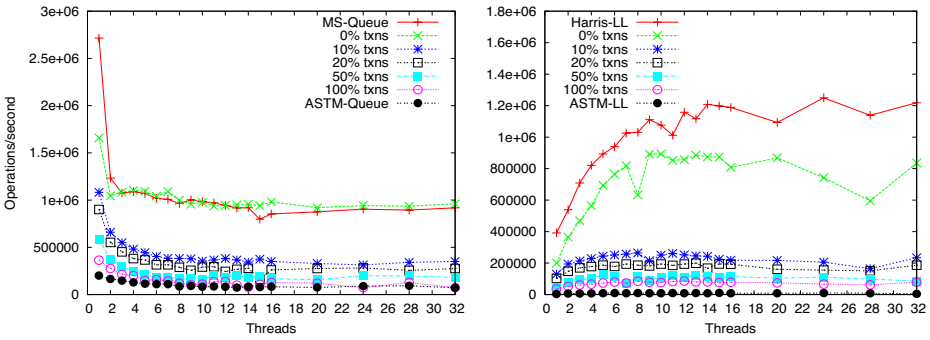

Fig. 1. Performance of transaction safe nonblocking objects with varying percentage of transactional and nontransactional invocations (50% inserts and 50% deletes). Experiments on a 16-processor 6800 SunFire cache coherent multiprocessor machine. Comparison with original nonblocking algorithms, and a natural transactional implementation.

additional programming effort, particularly if these primitives are supported in standard libraries.

Our preliminary implementation is in the context of the ASTM [3] system, where we extended the AtomicReference Java library class with a transaction aware version TxAtomicRef. We leveraged ASTM's transactional metadata structure (which consists of an indirection object called the *locator* that determines the current consistent version of the data, and its current writer transaction) to represent speculative values of TxAtomicRefs.

We implemented several nonblocking algorithms using TxAtomicRef including Michael and Scott's lock-free queue [4] and Harris' lock-free linked list [1] (results in Figure 1). In all cases we simply replaced the AtomicReferences in the original algorithms with TxAtomicRefs in our constructions. Our results suggest that while transaction safety makes nonblocking data structures somewhat slower, the resulting constructs interoperate smoothly with transactions, and can significantly outperform the natural "fully transactional" alternatives.

References

[1] T. L. Harris. A Pragmatic Implementation of Non-Blocking Linked-Lists. *15th Intl. Symp. on Distributed Computing*, Lisboa, Portugal, Oct. 2001.

[2] M. Herlihy and J. E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. *20th Intl. Symp. on Computer Architecture*, San Diego, CA, May 1993.

[3] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. *19th Intl. Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.

[4] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. *15th ACM Symp. on Principles of Distributed Computing*, Philadelphia, PA, May 1996.