Brief Announcement: Privatization Techniques for Software Transactional Memory*

Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott
Department of Computer Science, University of Rochester
{spear, vmarathe, luked, scott}@cs.rochester.edu

Categories and Subject Descriptors:
D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms: algorithms, performance

Keywords: synchronization, software transactional memory, privatization

1. INTRODUCTION

Transactional memory (TM) allows the programmer to encapsulate arbitrary memory operations into a single *transaction* that appears to be *atomic* and *isolated* from other transactions. In addition, TM systems must generally address the possibility that shared data may be accessed *outside* a transaction. The *strong isolation* model [1,3] guarantees that transactions are isolated from nontransactional ("naked") loads and stores, as well as other transactions. Due to the perceived high overheads for achieving strong isolation in software TM systems (STMs), the so-called *weak isolation* model, which guarantees isolation only among transactions, is also considered by some as an acceptable alternative.

Under weak isolation, a simple programming model is *single lock atomicity*: "a program executes *as if* all transactions were protected by a single, program-wide mutual exclusion lock" [3, pp. 20]. In traditional lock-based code, races between critical sections and naked loads and stores are usually considered program bugs. In a TM system with weak isolation, races between transactional and nontransactional accesses are likewise bugs.

The obvious way to cope with weak isolation is to ensure that no object is accessed by transactional and nontransactional code at the same time. That is, program logic must partition objects, at each point in time, into those that are *shared*, with access mediated by transactions, and those that are *private* to some thread. A transaction may modify shared state in such a way that (once the transaction commits) no future successful transaction will access a certain set of objects. This *privatization*, which serves to improve performance (by eliminating often significant overheads of transactional accesses) and to escape the prohibition against irreversible actions (e.g. I/O) within transactions, is semantically straightforward under the single lock atomicity model. Unfortunately, as we illustrate in Section 2, many STM implementations fail to guarantee correct program behavior in the presence of privatization.

*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.

PODC'07, August 12–15, 2007, Portland, Oregon, USA.
ACM 978-1-59593-616-5/07/0008.

Our work presents the first comprehensive study of privatization, including manifestations of (Section 2), and solutions to (Section 3) the problem (more detailed description appears in our technical report [4]). We describe tradeoffs among solutions for several different kinds of STMs, including systems based on indirection, undo logs, and redo logs.

2. THE PRIVATIZATION PROBLEM

Some recent work has recognized and addressed parts of the privatization problem, but none, to our knowledge, has examined all aspects of the problem. Larus and Rajwar [3, pp. 22-23] explain one half of the problem: how a privatizer might fail to see prior transactional updates, or might see updates made (temporarily) by a doomed transaction. Wang et al. [5, pp. 6-7] consider the possibility of erroneous behavior in a doomed transaction due to nontransactional updates made by a privatizing thread. Neither work discusses the other subproblem.

Transactions Cause Incorrect Private Reads.

In a typical STM system, nontransactional reads of privatized objects are not coordinated with concurrent transactions that may optimistically modify these privatized objects. As a result, private reads may return object values that are mutually inconsistent with other privately accessed objects. In redo-log based STMs, a transaction makes its speculative updates to a private write log. These updates are copied back, *non-atomically*, to the target objects when the transaction commits. As a result, a thread may privatize a group of objects (being modified by a committed transaction that is still doing its copyback) and read mutually inconsistent values of these objects in the nontransactional code.

In undo-log based STMs, a transaction makes its speculative updates directly to the target objects while maintaining a log of old values to be restored if the transaction subsequently aborts. Most STM implementations permit optimistic execution of transactions, wherein a doomed transaction may continue execution, but is guaranteed never to commit. In such an environment, a committing privatizer may implicitly abort a conflicting transaction due to its privatization operation (a write to some object/s that effectively privatizes a group of objects), and subsequent nontransacitonal code may read object values that were speculatively modified by the doomed transaction, which has yet to abort and roll back.

Indirection-based STMs have the same problem as undo-log based STMs: A writer transaction creates a clone of a target object and makes all modifications to the clone. The clone doesn't become the object's logical value until the writer commits, but may become visible to nontransactional code as soon as the writer acquires (locks) the object. As a result, a privatizing thread may access logically unreachable versions of objects speculatively modified by doomed transactions.

Private Writes Cause Incorrect Transactional Actions.

Nontransactional writes of privatized objects may not be coordinated with concurrent transactions that optimistically access these privatized objects. In an undo-log or redo-log based STM, *post-validation* (verification that transactional metadata of the object being accessed is mutually consistent with all prior accesses of objects in the transaction) on every read of an object does not help the transactional runtime detect that the object was modified by a private (nontransactional) write. This is because the private write does not modify object metadata. As a result, the transaction may read mutually inconsistent data, although corresponding transactional metadata is still consistent.

An indirection-based STM's transactions typically assume that readable objects are immutable, since modifications are always made to clones. However, nontransactional writes happen "in-place" on current logical versions of objects, not their clones. As a result, a transactional read may return the value of a private write, thus leading to the transaction observing inconsistent data.

A private write on an object by a thread may also be lost if a doomed transaction writes to that object. The doomed transaction, during its roll back, overwrites the private updates made by the privatizing thread to restore a stale value in the object.

3. PRIVATIZATION SOLUTIONS

We have considered several solutions to privatization. Due to space restrictions we overview the most interesting ones here (see our technical report [4] for a comprehensive discussion).

Explicit Fences.

Fence operations force a privatizing thread to wait until it is safe to continue. There are several possible implementations.

We can trivially ensure privatization safety by waiting, at the privatizer's commit point, for all active transactions to commit or abort, and finish any necessary cleanup – the *transactional fence*.

A transactional fence induces unnecessary delays for non-overlapping transactions. Note that a privatizer is guaranteed not to interfere with transactions that have started validation after the privatizer commits. We use the term *validation fence* to describe an operation wherein a privatizer waits for all threads to either be outside a transaction, or begin validation of the current transaction. Latency of a validation fence is expected to be much lower than a transactional fence, particularly in case of long transactions. However, the validation fence does not guarantee post- abort or commit cleanup of all objects that may be privately accessed. To prevent incorrect behavior of nontransactional code, the privatizing thread must inspect (and possibly clean up) the metadata associated with an object at the time of the first private access. This imposes overheads on nontransactional code.

The best features of the validation and transactional fences can be combined in STMs that employ global timestamp based transaction consistency checks [2, 5]. We call this a *global timestamp fence*. If every transaction T_k commits at a unique time C_k (as indicated by a global clock), then the timestamp fence can be implemented by maintaining a global list of active transactions and a global indication of the time at which the most recent transaction committed. During execution, T_k validates when it observes a new value for the last commit time C. It then writes the value C into its slot in the global list. It removes itself from the list after completing any post-commit or post-abort cleanup. To perform a fence, T_k waits until every entry in the list carries a timestamp $\geq C_k$. In systems without a global timestamp, the effect of a timestamp fence can be achieved with two consecutive validation fences.

Nonblocking Privatization.

Fence-based privatization techniques are fundamentally blocking. To eliminate blocking, the runtime must abandon fences. As a result, nontransactional code must inspect object metadata, not just on the first post-privatization access to each object, but for all accesses to each object. This is because a concurrent doomed transaction may modify a privatized object after the object was accessed by the privatizer. The doomed transaction does not identify that it is doomed and thus may make a privatized object inconsistent. Without metadata inspection, the privatizer might then access the object in its inconsistent state.

On the other side of the interaction, a doomed transaction may observe private writes performed by nontransactional code, causing erroneous behavior. Our solution is to perform full validation (of the entire read/write set of the transaction) whenever a privatizing thread commits. A privatizer increments a global *privatization counter* (pcount) after committing. During post-validation of every shared access, transactions verify that pcount has not changed. Transactions perform full validation whenever pcount changes. Furthermore, threads must poll pcount both before and after full validation, and must revalidate if pcount changes. Our observation is that if pcount changed, a privatizer must have committed during the validation, and the validating transaction must revalidate to ensure that there was no conflict.

While many overheads of nonblocking privatization can be mitigated through API or compiler support, the cost of contention on a global shared counter and corresponding full validation of all concurrent transactions is unfortunate. There is however an elegant simplification to STMs that employ global timestamps. When a privatizer commits, all objects written by the privatizer *within the transaction* are given the timestamp as their version number. We propose that the privatizer write the same timestamp as the version of any object it accesses outside the transaction. This ensures that no concurrent transaction will access these objects. The privatizer must of course clean up any object that is owned by another transaction. With this mechanism, no pcount, and hence no superfluous validation of concurrent transactions, is needed.

In the longer version of this paper [4] we present a preliminary performance evaluation of the different privatization techniques, as well as a programmer-centric taxonomy for privatization-safe STMs that embodies a tradeoff between programming complexity and performance.

4. REFERENCES

[1] C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *ACM SIGARCH Computer Architecture News*, 5(2), Nov. 2006.

[2] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, Sep. 2006.

[3] J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2007.

[4] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization Techniques for Software Transactional Memory. Tr 915, Dept. of Computer Science, Univ. of Rochester, Feb. 2007.

[5] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *Proc. of the Intl. Symp. on Code Generation and Optimization*, Mar. 2007.