Composable Transactional Objects: A Position Paper

Maurice Herlihy[1] and Eric Koskinen[2]

[1] Brown University, Providence, RI, USA
[2] New York University, New York, NY, USA

Abstract. Memory transactions provide programmers with a convenient abstraction for concurrent programs: a keyword (such as atomic) designating a region of code that appears, from the perspective of concurrent threads, to execute atomically. Unfortunately, existing implementations in the form of software transactional memory (STM) are often ineffective due to their monolithic nature: every single read or write access is automatically tracked and recorded.

In this statement, we advocate a transactional model of programming without a heavyweight software transactional memory, and describe some related, open research challenges. We suggest that a model based on *persistent data structures* could permit a variety of transactional algorithms to coexist in a library of *composable transactional objects*. Applications are constructed by snapping these objects together to form atomic transactions, in much the same way that today's Java programmers compose their applications from libraries such as java.util.concurrent.

We report preliminary results developing this library in ScalaSTM, and discuss the challenges ahead.

Keywords: Composable transactional objects, transactional memory, persistent, multicore.

1 Introduction

Existing transactional memory systems (hardware [3,11,12], software [9,18,6], or hybrid [5,16]) detect conflicts at a *read-write* level: each transaction keeps track of a *read set*, the locations it read, and a *write set*, the locations it wrote. Two transactions are deemed to conflict if one's write set intersects the other's read or write set. The TM run-time typically intercepts all memory accesses, tracks each transaction's read and write sets, and delays or restarts transactions that encounter conflicts.

There is an increasing realization that tracking read-write conflicts is *inefficient*, because each and every memory access must be monitored for conflict and recorded for potential roll-back, and *ineffective*, because *false conflicts* frequently arise when read and write sets inadvertently intersect in a harmless way. For example, consider an object that generates unique identifiers. Logically, there is no reason that concurrent identifier requests should conflict. If the generator is

Z. Shao (Ed.): ESOP 2014, LNCS 8410, pp. 1–7, 2014.

implemented in a natural way as a counter, however, then today's STM systems will unnecessarily detect a conflict. Perhaps as a result, performance remains a barrier to widespread use of today's STMs.

In this position paper, we propose an alternative research direction, based on libraries of *composable transactional objects*, which we are currently building using ScalaSTM [1]. Our goal here is to outline a research vision, calling attention to open problems and new directions.

In this alternate direction, the unity of our library is not ensured by a monolithic STM but instead defined at a higher level: *persistent objects* [7]. Informally, this property ensures that one can reconstruct (some or all) earlier versions of the object even after it has been modified. The notion of persistent objects allows us to combine diverse transactional algorithms into composable objects that "snap together" to form atomic transactions. Our prior work on *transactional boosting* [10], is an example of how one might implement a composable transactional object, replacing bit-level read-write conflicts with a high-level notion of conflicts between non-commutative methods of abstract data types.

2 Overview

We advocate a move away from the pervasive notion that transactional synchronization must be done on the basis of *read-write conflicts*. Synchronization based entirely on read-write conflicts has three drawbacks: (*i*) it can limit concurrency through false conflicts, (*ii*) it can burden performance by instrumenting too many memory accesses, and (*iii*) it can hamper recovery by requiring bit-wise copying of large amounts of data.

Nonetheless, a library of transactional objects is only useful if the objects can be combined together. In this section we describe a new route toward a library of composable transactional objects via the notion of *persistent* objects. Let's begin with an example object.

Example Transactional Object. Consider the object in Fig. 1 that implements transactions via *boosting* [10]. This figure (see the original paper [10] for a more systematic explanation) shows part of the (Scala) code for a highly-concurrent transactional key-value map that provides put() and get() methods. The base object is the ConcurrentSkipListMap class from the java.util.concurrent library. For transactional synchronization, the key insight is that method calls for distinct keys commute, so concurrent transactions that operate on distinct keys can proceed in parallel, even if their underlying read and write sets conflict. In this code, transactional isolation is provided by our AbstractLock class, which associates each key value (via an internal hash table) with an abstract lock. Abstract locks are strict two-phase locks: each method call acquires the lock associated with its key (Line 6), to be released when the transaction commits or aborts (Line 7). If the transaction eventually aborts, the run time system is requested to restore the previous binding if there was one (Line 10), or to remove the new binding if there wasn't (Line 12). Finally, the new binding is placed in the map (Line 14).

```
1  import java.util.concurrent.ConcurrentSkipListMap
2  class BoostedSkipList[Key,Value] {
3    private val abstractLock = new AbstractLock()
4    private val map = new ConcurrentSkipListMap[Key, Value]()
5    def put(key: Key, value: Value, t = Transaction.current): Unit = {
6      abstractLock lock key
7      Transaction.onExit ( () => abstractLock unlock key )
8      if (map containsKey key) {
9        var oldValue = map.get(key)
10       Transaction.onAbort(() => map.put(key, oldValue))
11     } else {
12       Transaction.onAbort(() => map remove key)
13     }
14     map.put(key, value)
15   }
16    ...
17 }
```

Fig. 1. A boosted Concurrent Skip List

We have "boosted" a highly complex and highly optimized skip-list map implementation, written by someone else, from being thread-safe to transaction-safe, without rewriting a line of its code. Because the base ConcurrentSkipListMap class provides its own thread-level synchronization, it is safe for concurrent threads to make put() calls concurrently at Line 14. Moreover, there is no need for an underlying STM to intercept and track each low-level read and write access, nor to block or roll back transactions whose read and write sets overlap. Here, transaction recovery is implemented by logging and replaying inverse operations, potentially a much more compact and efficient means of recovery than the usual STM technique of manipulating large, bit-level *before* and *after* images. Deadlocks are detected and resolved using the *Dreadlocks* deadlock detection algorithm [14] developed for this purpose. Finally, this boosted implementation satisfies *opacity* [8], a correctness condition that ensures that all transactions, even those doomed to abort, observe a consistent memory state.

3 Persistent Data Structures

Boosting marks an escape from the monolithic approach present in today's STMs. While there is a substantial performance improvement, we have lost the uniformity of a monolithic STM. It is natural to wonder: how can such transactional objects interoperate with other objects that, themselves, may utilize (possibly different) transactional algorithms?

We argue that we can elevate the common conceptual framework that unifies diverse transactional algorithms. A boosted object can coexist with a transactional object built, for example, in a speculative manner (as discussed next).

And so on. This is what Java programmers, who today combine myriad lock/lock-free java.util.concurrent objects, would expect of a library of transactional objects. We argue that this can be done with objects that are persistent:

Definition 1 (Persistent Object [7]). *A mutable data object is* persistent *if one can reconstruct earlier versions even after the object has been modified. It is said to be* partially persistent *if only some versions can be reconstructed, and it is* confluently persistent *if new versions created by concurrent activities can be merged in a meaningful way.*

Informally, persistent objects allow us to scroll backwards and forwards through time, giving us a great deal of flexibility at run-time to serialize concurrent object operations. Of course a completely persistent object is impractical. So this leads us to research questions such as: *Which* earlier versions must persist? *For how long* must they persist?

Let's look at an example. Here is how one can make a boosted object be persistent. If the object retains the undo logs of committed transactions, then any earlier version can be reconstructed by cloning the base object, and replaying the undo log back to the desired version.

Our use of persistent objects as a basis for both transactional synchronization and semantics is an attempt to combine the well-known benefits of functional programming with the unavoidable need for high-level mutable state, much in the spirit of our earlier work on transactional Haskell [9].

4 Optimism

In boosting, transactions apply method calls directly to the base object, relying on an operation-based undo log to roll back failed transactions. In this way, synchronization in boosting is *pessimistic*, because transactions check for conflicts before calling a method. An alternative is *optimistic* (or *speculative*) synchronization, where transactions check for conflicts only at the end. (Checking for conflicts is often called *validation*.) Many STM systems (for example, TL2 [6]) operate this way: updates to shared memory are deferred until commit. Optimistic synchronization can reduce costs if conflicts are sufficiently rare.

Here is another scenario where deferred updates might be attractive. In a non-uniform memory access (NUMA) architecture, threads can access local memory quickly, and remote memory more slowly. In such a situation, each thread might operate on its own local copy of the base object. When it commits after validation, it propagates its changes (in the form of an operation-based redo log) to the other threads. The Barrelfish [2] operating system is organized around a similar philosophy.

Optimistic synchronization involves objects that are *confluently persistent* [7]: new object versions can be created by concurrent activities as long as those versions can be merged in a meaningful way. Usually, operations can be merged as long as they commute, but weaker properties, involving left- and right-movers, can also be used [15].

This move toward composable transactional objects enables us to incorporate other transactional features such as checkpoints and nested transactions. We can even model *dependent* transactions [17], where one transaction releases its results to another before committing, and the second transaction's commit depends on the first's.

5 Preliminary Results and the Road Ahead

We have embodied our ideas in library of composable transactional objects, implemented in ScalaSTM. Our implementation replaces the existing heavyweight run-time that mediates all transactional memory interactions with a much less obtrusive structure. Our system provides only the following services:

- onCommit() registers a closure to be called when a top-level transaction commits. Closures are called in first-in-first-out order, useful for redo logs.
- onAbort() registers a closure to be called when a transaction (nested or top-level) aborts. Closures are called in last-in-first-out order, useful for undo logs.
- onExit() registers a closure to be called when a top-level transaction commits or aborts, useful for releasing abstract locks, certain kinds of I/O, and memory management.
- onValidate() registers a Boolean-valued closure to be called before a top-level transaction commits or aborts. A transaction commits only if all such return values are *true*. This service is useful for speculative synchronization.

Versioning. At the implementation level, object versions are indexed by transaction identifiers. At all times, there is a unique system-wide identifier for the latest committed transaction, which indexes the latest committed state for each object. Operations of composable transactional objects take a transaction identifier as a default argument, with the currently executing transaction as the default. Objects are confluently persistent in the sense that they can permit concurrent method calls to the committed version, provided the object implementation is capable of merging them, based on commutativity or other type-specific properties. When a thread commits a transaction, it installs that transaction as the latest committed transaction, when it aborts, it discards that transaction and the versions it indexes do not become accessible to the other threads. A long read-only transaction is one that executes under a committed transaction, running against a set of object versions "frozen" at that time. (Not all objects will provide access to older versions.)

Challenges. Our next step is to finish a comprehensive implementation of composable transactional objects with a wide range of transactional algorithms. There are then some open research challenges, including:

1. Investigating trade-offs between granularity and performance in data structure design, and port benchmarks such as STAMP [4] to ScalaSTM.

2. Investigating how special support can be added to aid long-running (in particular, read-only) transactions.
3. Exploring other novel control structures, such as the retry construct for conditional transactional synchronization, and the orElse construct for composing conditional synchronization (as introduced in Transactional Haskell [9]). Elsewhere [13], we described how boosting can be extended to support these and other useful control structures, but a more general approach to composable transactional objects will require rethinking and extending these mechanisms.
4. Exploiting hardware transactions, of the kind recently provided by Intel Haswell [12] and soon to be provided by the IBM Power architecture [3].
5. Developing accessible verification techniques to ensuring the correctness of these objects which we believe will be used widely.

References

1. ScalaSTM, http://nbronson.github.io/scala-stm/
2. Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., Singhania, A.: The multikernel: a new os architecture for scalable multicore systems. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP 2009, pp. 29–44. ACM, New York (2009)
3. Cain, H.W., Michael, M.M., Frey, B., May, C., Williams, D., Le, H.: Robust architectural support for transactional memory in the power architecture. In: Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA 2013, pp. 225–236. ACM, New York (2013)
4. Cao Minh, C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., Olukotun, K.: An effective hybrid transactional memory system with strong isolation guarantees. In: Proceedings of the 34th Annual International Symposium on Computer Architecture, ISCA 2007 (June 2007)
5. Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybrid transactional memory. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII), pp. 336–346. ACM Press, New York (2006)
6. Dice, D., Shalev, O., Shavit, N.N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
7. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. J. Comput. Syst. Sci. 38(1), 86–124 (1989)
8. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, PPoPP 2008, pp. 175–184. ACM, New York (2008)
9. Harris, T., Marlow, S., Peyton-Jones, S.L., Herlihy, M.: Composable memory transactions. Commun. ACM 51(8), 91–100 (2008)
10. Herlihy, M., Koskinen, E.: Transactional boosting: a methodology for highly-concurrent transactional objects. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2008, pp. 207–216. ACM, New York (2008)
11. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, ISCA 1993, pp. 289–300. ACM Press (1993)

12. Intel Corporation. Transactional Synchronization in Haswell (September 8, 2012), http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/ (retrieved from)
13. Koskinen, E., Herlihy, M.: Checkpoints and continuations instead of nested transactions. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2008, pp. 160–168. ACM, New York (2008)
14. Koskinen, E., Herlihy, M.: Dreadlocks: efficient deadlock detection. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2008, pp. 297–303. ACM, New York (2008)
15. Koskinen, E., Parkinson, M., Herlihy, M.: Coarse-grained transactions. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, pp. 19–30. ACM, New York (2010)
16. Moravan, M.J., Bobba, J., Moore, K.E., Yen, L., Hill, M.D., Liblit, B., Swift, M.M., Wood, D.A.: Supporting nested transactional memory in logtm. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII), pp. 359–370. ACM Press, New York (2006)
17. Ramadan, H.E., Roy, I., Herlihy, M., Witchel, E.: Committing conflicting transactions in an stm. In: PPOPP, pp. 163–172 (2009)
18. Saha, B., Adl-Tabatabai, A.-R., Hudson, R.L., Minh, C.C., Hertzberg, B.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2006, pp. 187–197. ACM, New York (2006)