NVMW 2017

Composing Durable Data Structures

Joseph Izraelevitz
University of Rochester
Rochester, NY
jhi1@cs.rochester.edu

Virendra J. Marathe
Oracle Labs
Burlington, MA
virendra.marathe@oracle.com

Michael Scott
University of Rochester
Rochester, NY
scott@cs.rochester.edu

***Abstract*—This paper presents techniques for composing persistent data structure operations on machines with nonvolatile byte addressable memory. The techniques are applicable to a wide class of nonblocking algorithms.**

I. Introduction

The advent of *byte addressable*, *nonvolatile* main memory technologies (such as PCM or STT-MRAM) will affect the way we build software that manages persistent data, as this technology enables durable data storage in main memory.

Programmers may wish to move existing in-memory data structures to nonvolatile storage to allow them to persist from one run to another, and to survive processor crashes. Several challenges, however, make the move more difficult than it might at first appear.

Processor caches and registers are expected to remain volatile (and the data in them *transient*) for the foreseeable future: a power failure means that nonvolatile RAM state remains but cache and register state is lost. And since cache lines may be written back to memory in arbitrary order, the simple load/store interface is not sufficient to ensure the consistency of persistent state; programs must take steps to control the order of writes-back (in our work we rely on Intel's clflush instruction, which forces a cache line into persistence and blocks until the write-back completes).

Several groups have developed concurrent data structures for a machine model with nonvolatile RAM but volatile caches. Automated algorithms also exist to transform nonblocking transient data objects into persistent ones [1]. Additional theoretical work has discussed correctness criteria for concurrent data structures. In particular, most published designs provide *durable linearizability* [1], an extension of the traditional linearizability correctness criterion [2] into persistence. Informally, a durably linearizable object ensures that each of its methods, between its invocation and return, (1) becomes visible to other threads atomically and (2) reaches persistence in the same order that it became visible.

Looking beyond individual objects, we should like to be able to *compose* operations on pre-existing durably linearizable objects into into larger failure-atomic sections (i.e., transactions). Such composability might be seen as an extension of *transactional boosting* [3], which allows operations on linearizable data structures (at least those that meet certain interface criteria) to be treated as primitive operations within larger atomic transactions.

In this extended abstract, we discuss additional interface requirements for durably linearizable data structures in order for them to be atomically composable. We also present a simple, universal, lock-free construction, which we call the *chronicle*, for building data structures that meet these requirements.

II. Composition

Composition is a hallmark of transactional systems, allowing a set of nested actions to have "all-or-nothing" semantics. The default implementation arranges for all operations to share a common log of writes (and reads, for transactions that provide isolation), which commit or abort together. Unfortunately, this implementation imposes overhead on every memory access, and leads to unnecessary serialization when operations that "should" commute cannot due to conflicting accesses to some individual memory location internally.

Boosting addresses both of these problems by allowing operations on black-box concurrent objects to serve as "primitives"—analogues of read and write—from the perspective of the transactional system. In a system based on UNDO logs, memory updates are made "in place" and *inverse* operations are entered in an UNDO log. For a write, the inverse is a write of the previous value. For a higher-level operation, the inverse depends on the semantics of the object (a push's inverse is a pop). In the event of a transaction abort, the log is played in reverse order, undoing both writes and higher level operations using their inverses. For concurrency control, *semantic locks* are used to prevent conflicts between operations that do not commute (e.g., puts to different keys commute, but puts to the same key do not; transactions that access disjoint sets of keys can run concurrently).

We aim to extend the boosting of linearizable objects in (transient) transactional memory so that it works for durably linearizable objects in persistent transactional memory. To do so, we must overcome a pair of challenges introduced by the possibility of crashes. First, transactional boosting implicitly assumes that a call to a boosted operation will return in bounded time, having linearized (appeared to happen instantaneously) sometime in between. While we can assume that a durably linearizable object will always be consistent in the wake of a crash (as if any interrupted operation had either completed or not started), we need for composition to be able to *tell* whether it has happened (so we know whether to undo or redo it as part of a larger operation). Second, transactional boosting implicitly assumes that we can use the

return value of an operation to determine the proper undo operation. For composition in a durably linearizable system, we need to ensure that the return value has persisted—so that, for example, we know that the inverse of S.pop() is S.push(v), where v is the value returned by the pop.

III. Query-Based Logging

One method of durable boosting employs what we call "query-based logging," a technique applicable to both UNDO and JUSTDO logging [4]. In our design, the boosted durable data structure is responsible for maintaining sufficient information about interrupted operations to ensure both that their inverses can be computed and that they are executed only once. An interrupted transaction can *query* the data structure after the crash using a unique ID to gather this information.

The query interface is designed as follows. All the normal exported methods of a boostable data structure take a unique ID for every invocation (e.g., a thread ID concatenated with a thread-local counter). There also exists a *query method*, which takes a unique ID as argument and returns either NULL, indicating that the operation never completed and never will, or a struct containing the operation's invoked function, corresponding arguments, and return value.

Boosting using query-based UNDO logging is straightforward. The transaction is executed sequentially, and acquires the appropriate read, write, and semantic locks as needed. Before a boosted operation, we log our intended operation in the UNDO log. After the operation returns, we mark the operation completed in the UNDO log, and, if appropriate, record its return value. If the operation is interrupted, we can use the query interface to determine if the operation completed and what its return value would be. Using this information, we can complete (or ignore) the UNDO entry, then roll back the transaction in reverse using the normal UNDO protocol and each operation's inverse. JUSTDO logging works similarly, but rolls forward from the interrupted operation.

A. The Chronicle

To facilitate the use of query-based logging, we present a lock-free construction, called the *chronicle*, that creates a queryable, durably linearizable version of any data structure with the property that each method linearizes at one of a statically known set of compare-and-swap (CAS) instructions, each of which operates on a statically known location. This property is satisfied by, for example, any object emerging from Herlihy's classic nonblocking constructions [5]. In our construction, each CAS-ed location is modified indirectly through a State object. Instead of using a CAS to modify the original location, an operation creates a new global State object and appends it to the previous version. By ensuring that all previous States have been written to persistent storage before appending the new State, we can ensure that all previous operations have linearized and persisted. By attaching all method call data to the State object associated with its linearization point, we can always determine the progress of any ongoing operation.

```
1  class Node{
2   Object val;
3   // the stored object
4   Node* down;
5   // the next node down
6  };
7  class State{
8   State* next;
9   // the next State in
10  // the chronicle
11  Node* head;
12  // the head Node
13  int method;
14  // method invoked
15  int uid;
16  // a unique id for op
17  void* ret;
18  // return value of op
19 };
20 class Stack{
21  State* chronicle;
22  Stack(){chronicle=
23    new State(NULL,NULL,
24    INIT,0,NULL);}
25 };
26 State* Stack::flushChronicle
27 (State* fromHereForward){
28  State* s = fromHereForward;
29  while (s→next≠ NULL){
30   clflush(s);
31   s = s→next;
32  }
33  State* realState = s;
34  clflush(realState);
35  // now chronicle is
36  // entirely flushed
37  return realState;
38 }
```

```
39 Object Stack::pop(int uid){
40  State* s = chronicle;
41  while(true){
42   s = flushChronicle(s);
43   Object x = h→head→val;
44   Node n = s→head→down;
45   s_new =
46    new State(NULL,n,POP,uid,x);
47   // append new State to the
48   // stack and chronicle
49   if(CAS(&s→next,NULL,s_new)){
50    clflush(s);
51    // flush CAS to s→next
52    return x;
53   }
54  }
55 }
56
57 int Stack::push
58  (Object x, int uid){
59  State* s = chronicle;
60  while(true){
61   s = flushChronicle(s);
62   Node* n = new Node(x,s→head);
63   clflush(n)
64   s_new =
65    new State(NULL,n,
66    PUSH,uid,SUCCESS);
67   clflush(s_new);
68   // append new State to the
69   // stack and chronicle
70   if(CAS(&s→next,NULL,s_new)){
71    clflush(s);
72    // flush change to s→next
73    return SUCCESS;
74   }
75  }
76 }
```

Fig. 1. Treiber Stack Chronicle Implementation

To demonstrate the utility of the chronicle, Fig. 1 presents a variant of the non-blocking Treiber stack [6]. Like the original, this version is linearizable. Unlike the original, it provides durable linearizability and a queryable interface. While the version here flushes the entire chronicle on every operation, simple optimizations can be used to flush only the incremental updates and to garbage collect old entries.

References

[1] J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of persistent memory objects under a full-system-crash failure model," in *Proc. of the 30th Intl. Conf. on Distributed Computing*, ser. DISC '16, Paris, France, 2016, pp. 313–327.

[2] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[3] M. Herlihy and E. Koskinen, "Transactional boosting: A methodology for highly-concurrent transactional objects," in *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, ser. PPoPP '08, Salt Lake City, UT, USA, 2008, pp. 207–216.

[4] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via JUSTDO logging," in *Proc. of the 21st Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XXI, Atlanta, GA, USA, 2016.

[5] M. P. Herlihy, "A methodology for implementing highly concurrent data objects," *ACM Trans. on Programming Languages and Systems*, vol. 15, no. 5, pp. 745–770, Nov. 1993.

[6] R. K. Treiber, "Systems programming: Coping with parallelism," IBM Almaden Research Center, Tech. Rep. RJ 5118, Apr. 1986.