Prior Art: Data Structures and Transactional Memory

Prior solutions for crash consistency on NVM have focused on two major areas. **Data structures** designed for NVM ensure that their metadata and contents are consistent after a crash, and operations become persistent in a well-regulated manner (e.g., they meet the correctness condition **durable linearizability** [1]). In contrast, transactional systems guarantee that all changes from **failure atomic sections** of code (e.g., transactions) are entirely visible or entirely dropped after a crash.

**Our Goal: Composition**

This work investigates composing operations on durably linearizable data structures into larger failure atomic sections (e.g., transactions). This goal can be seen as an extension of **transactional boosting**, a technique used in traditional (transient) transactional memory.

**Transactional Boosting**

Boosting for transactional memory eschews traditional read/write logging for **semantic logging** on concurrent data structures [2]. In an UNDO system, data structure updates are done in place, and inverse operations are written in the log (the inverse of a `pop()` is a `push()` of the return value). On abort, the log is played backwards, inverting the data structure operations. Semantic locks are used to prevent conflicts between operations that do not commute.

**Problem: Boosting in Persistence**

Transactional boosting implicitly assumes that a call to a boosted operation will return in bounded time. While we can assume that the data structure will always be consistent in the wake of a crash (as if any interrupted operation had either completed or not started), for composition we need to be able to tell whether the operation has happened (so we know whether to undo or redo it as part of a larger operation). Second, transactional boosting assumes that we can use the return value of an operation to determine the proper UNDO operation. For composition in persistence, we need to ensure that the return value has persisted so we can calculate the inverse.

**Solution: Query-Based Logging**

**Query-based logging** enables composition of persistent data structures into failure atomic sections.

In query-based logging, the data structures are responsible for persisting sufficient state to identify if an operation has occurred and what its return value is. Transactions can query the data structure after a crash for this information.

```plaintext
Foreach op in transaction{
    1) Acquire semantic lock
    2) Store op’s unique ID in log
    3) Perform op on structure
    4) Mark op complete in log w/ return value
}
Release all transaction locks

On abort:
Query incomplete operation with ID to get status and return value. Play log backwards.
```

**Building Queryable Data Structures**

The **chronicle** is a lock-free universal construction for building queryable data structures, suitable for query-based logging. The chronicle maintains old versions of the data structure for answering queries and ensures that old versions become persistent before new versions are created.

At a high level, the chronicle contains a list of **State** objects which maintain pointers to the structure root. We demonstrate with a **Stack**, which starts empty and transient (red).

Before beginning a new operation, a thread ensures that all previous **State** objects are persistent (blue) using clflush.

Having persisted previous states, the thread creates a new head of the stack and **State** object, persisting both.

The new **State** is appended to the list using a CAS. Subsequent updates will force this CAS into persistence before adding new **States** to the chronicle.

---
