Transaction Local Aliasing in Storage Class Memory

Ellis Giles
Rice University
Email: erg@rice.edu

Kshitij Doshi
Intel Corporation
Email: kshitij.a.doshi@intel.com

Peter Varman
Rice University
Email: pjv@rice.edu

This paper describes a lightweight software library to solve the challenges [6], [3], [1], [5], [2] of programming storage class memory (SCM). It provides primitives to demarcate failure-atomic code regions. SCM loads and stores within each demarcated code region (called a “wrap”) are routed through the library, which buffers updates and transmits them to SCM locations asynchronously while allowing their speedy propagation from writers to readers through CPU caches.

Algorithm 1 shows an example (on the left) in which three variables need to be either all updated or none updated in persistent memory. To achieve failure atomicity of such a sequence two complementary problems need to be solved.

Algorithm 1: Programmer annotated failure atomic region

```
// x, y and z are persistent variables
atomic_begin | id = OpenWrap();
x = 1;       | wrapStore(id, &x, 1);
.....      |
y = 2;       | wrapStore(id, &y, 2);
.....      |
z = x;       | temp = wrapLoad(&x);
.....      | wrapStore(id, &z, temp);
atomic_end   | CloseWrap(id);
```

The *durability* problem is ensuring that a store made to a persistent memory address has actually been made persistent on the non-volatile medium. To provide this guarantee, manufacturers are providing a persistence commit instruction (e.g., x86 PCOMMIT) with which software can achieve confirmation that previously flushed cache lines have reached a power-safe domain. A complementary problem is to ensure that uncontrolled background evictions by the cache controller *do not* update persistent memory in a way that can violate failure atomicity. The paper extends an approach called *SoftWrAP* [4] (for software based write-aside persistence) to perform failure atomic multi-element writes to SCM efficiently from concurrency-safe regions of code.

SoftWrAP Approach: The basic idea in SoftWrAP [4] is to simultaneously propagate updates made within an atomic region along two paths: a foreground path through the cache hierarchy that is used for value communication within and between wraps, and an asynchronous background path to SCM for recovery logging. In [4], a shared, volatile alias table containing the most recent values is used for value communication between writers and readers. New values are entered into the alias table while being streamed to a SCM based log. Post-transactional values are transferred to SCM from the alias table and space for the associated entries and log are removed. By creating these two paths, SoftWrAP decouples transaction value communication from recovery logging.

The decoupling of concurrency control (or transaction isolation) from failure atomicity allows persistence to be added flexibly to code that is already multi-threading safe. For

the common case of multi-threading safe or strictly isolated transactions, this paper further decouples the reclamation of the aliasing structure from the retirement of transaction logs, and makes post-transactional values available immediately to subsequent transactions without aliasing. We call this design SoftWrAP-LAT, for SoftWrAP with a Local Aliasing Table.

Using Algorithm 1, we first describe how [4] achieves failure safe updates in SCM, and then how SoftWrAP-LAT simplifies the transactional persistence operation. The atomic persistence region starts with *atomic_begin* which translates into a call to the *OpenWrap* library function as shown on the right in Algorithm 1. *OpenWrap* starts a log record for the transaction in SCM. Each store, such as $x = 1$, translates into a library call to *wrapStore*, which updates an entry for x , designated by x_a in a shared alias table in DRAM, allocating space if it does not exist, with the SCM address of x , its size and new value. Similarly it appends an entry to the log with x ’s SCM address, its new value (1), and size. A load of a wrapped variable, such as $z = x$ translates into a library call to *wrapLoad*, which uses the alias of x , x_a , to return the value of 1 from the shared alias table for use in the assignment. When the transaction closes with *atomic_end*, the *CloseWrap* library function performs a streaming flush of the log record with an end-of-record marker, and performs a PCOMMIT instruction to ensure the log record is safely in SCM. Subsequent transactions that need to read values of variables x , y , z modified in this atomic persistent region use *wrapLoads* to get them from the shared alias table. In the background, post transactional values of persistent variables are copied into SCM home locations and alias table entries are removed in lock step with trimming of the log records. A double buffered, multi-state shared alias table provides for lock-free reclamation of space [4].

For entire persistence transactions which are strictly isolated from one another, as with serialization schemes such as locks, a shared alias table is not necessary. Instead, a local aliasing scope suffices to forward values from writers to readers. The SoftWrAP-LAT scheme introduced in this paper is shown in Algorithm 2; its difference from a shared aliasing scheme of [4], is discussed. As shown in Figure 1, each persistence transaction maintains a private (local) alias table that is allocated in DRAM at the time of *OpenWrap* when the transaction also allocates a log record. The library calls *wrapStore* and *wrapLoad* operate on aliased entries x_a , y_a , and z_a as in [4], except that the aliasing is performed through *Transaction Local Aliasing*, since strict isolation removes any overlap among variables written in one transaction and concurrently read in another. On *CloseWrap*, each persistence transaction streams the updated values from its local alias table to the cache hierarchy. The updated values may asynchronously move to SCM and are available in place (at their SCM addresses) without aliasing, for subsequent transactions.

Fig. 1: Local Aliasing in SoftWrAP

To expedite trimming of log records, updates are flushed (using CLWB) and durably fenced (using PCOMMIT); however, it is possible to batch the fencing for data value updates for multiple transactions as an optimization.

The SoftWrAP-LAT approach can be also used when applications can tolerate unprotected reads, i.e. eager reads of variables (not yet durably committed) that produce stale values. Also, SoftWrAP-LAT can make streaming of log and data values to SCM efficient with AVX VSCATTER operations, akin to SoftWrAP [4]. Further, a compiler can be used to maximize static aliasing at compile time since SoftWrAP-LAT aliased values do not exist beyond each wrap.

Algorithm 2: Implementation of WrAP API

OpenWrap():

Allocate private Alias Table in DRAM;
Allocate Log Bucket in NVM;

WrapStore(id, addr, val):

Update or Add to Alias Table entry *addr* with *val*;
Write $\langle addr, val \rangle$ to Log Bucket [*id*] in NVM;

WrapLoad(id, addr):

Return *val* from Alias Table entry *addr* if present;

CloseWrap():

PCOMMIT to make log records durable;
For each entry in the Alias Table:
Non-blocking cached store *val* to SCM *addr*; CLWB *addr*;
Reclaim Local Alias Table and Commit Transaction;

Evaluation: Our evaluation is performed on an Intel(R) Xeon(R) CPU E5-2640 at 2.50 GHz with 64 GB of DDR3 system memory running RHEL Server 6.5 and built with GCC 4.4.7. We compare *SoftWrAP-LAT* with a 512 entry Local Alias Table (LAT) against an *UndoLog* based approach to persistence, which uses a log to save old values by committing synchronously to SCM before new values are written.

We first insert 20K random elements into a wrapped STX B+Tree data structure backed by an emulated SCM area in DRAM. Each insert into the B+Tree requires multiple other operations in the data structure that must also be performed atomically to preserve the integrity of the tree. On average over 60 reads and almost 20 writes are required to be performed atomically on the durable STX B+Tree of this size per insert of a single element. Figure 2a shows the relationship between the average response time to insert a new element into the tree for a varying insert request rate with a request buffer of size

Fig. 2: Transaction of inserting an element into a B+Tree. Response time for various arrival rates and maximum throughput are shown.

10. SoftWrAP has the fastest response time since UndoLog must perform a synchronous copy of the old data before the new data can be written. SoftWrAP only needs to persist new data out of the LAT to SCM before proceeding. Additionally, Figure 2b shows that SoftWrAP has a maximum sustained throughput almost twice that of the UndoLog approach, due to the UndoLog synchronous copies.

Related Work: Existing software solutions (e.g. [6], [1]) combine concurrency control with persistence in an integrated framework. Mnemosyne [6] uses software transactional memory (STM) based interception of all writes and reads within a transaction, and uses internal copying and logging to achieve both concurrency control and atomicity. However, it constrains the programmer to a single concurrency control model (TM), and is difficult to fit to legacy software applications, which support different isolation models or employ lock-based concurrency control. ATLAS [1] uses a compiler pass to automatically generate transactional regions for atomic writes utilizing a synchronous undo log, merging the providing of atomicity to a group of operations with controlling concurrent accesses among them. Analysis of consistency models for persistent memory was considered in [5]. An early presentation of SoftWrAP technique [4] was introduced in [3], and a similar approach for supporting transactional writes to persistent memory is described in REWIND [2], which offers a more elaborate alternative for in-place updates with multilayered recovery aided by an Atomic Doubly Linked List.

*Supported by NSF Grant CCF 1439075 and Intel SSG.

REFERENCES

- [1] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM OOPSLA '14*, pages 433–452, 2014.
- [2] A. Chatzistergiou, M. Cintra, and S. D. Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *VLDB'15*, 8(5):497–508, 2015.
- [3] E. Giles, K. Doshi, and P. Varman. Software support for atomicity and persistence in non-volatile memory. In *Memory Architecture and Organization Workshop (MeAOW'13)*, October 2013.
- [4] E. Giles, K. Doshi, and P. Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *31st International Conference on Massive Storage Systems and Technology*, 2015.
- [5] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *ISCA'14*, pages 265–276, 2014.
- [6] H. Volos, A. J. Tack, and M. Swift. Mnemosyne: Lightweight persistent memory. In *ACM ASPLOS'11*, pages 91–104, 2011.