Transactional Storage Class Memory

Ellis Giles
ACM Student Member
Rice University

Peter Varman
Advisor
Rice University

***Abstract*—Emerging Storage Class Memory, or SCM, technologies are promising both byte-addressability and persistence near DRAM speeds operating on the main memory bus. This high-speed, byte-addressable persistence will give rise to new applications that no longer have to rely on slow, block based storage devices and to serialize data for persistence. However, when writing values into a persistent memory tier, programmers are faced with a dual edged problem of how to catch spurious cache evictions while atomically grouping stores to manage consistency guarantees in case of failure.**

Multi-core processors and concurrent programming techniques are crucial for High Performance Computing applications. Consistency for byte-addressable persistent data coupled with multiple threads will be challenging for future applications. Both Hardware and Software Transactional Memory attempt to solve concurrency challenges but do not address high performance durability. On the other extreme, logging techniques that address durability of data structures in view of failure are not conducive for concurrent high performance applications.

This paper identifies and evaluates areas in both Hardware and Software Transactional Memory approaches that can be extended to provide durability on Storage Class Memories with little burden on performance. Several of these approaches are implemented as a library called Transactional Storage Class Memory or TSCM.

I. Introduction and Background

Storage Class Memory, or SCM, is an exciting new memory technology with the potential to replace hard drives and SSDs as it offers high-speed, byte addressable persistence on the main memory bus. With emerging SCM, applications can operate on persistent data directly in a byte-addressable fashion with plain loads and stores, but data can be corrupted by cache evictions or crashes. Therefore, programmers must ensure persistence consistency when writing to SCM.

Chip Multi-Processors and concurrent programming techniques are crucial for High Performance Computing. However, it is often difficult to program concurrent applications efficiently, with many programmers relying on the ease of global locks or coarse-grained locking techniques. Some languages, hardware, and tools introduce an atomic keyword, which attempts to perform fine-grained locking with the ease of programming coarse-grained locks. Hardware and Software Transactional Memory systems attempt to alleviate programmability difficulty by providing near-fine grained lock performance with ease of programming. HTM [2] is an easy to use method for lock-free synchronization supported for hardware. Intel uses a technique based on Speculative Lock Elision [3] called Intel Transactional Synchronization Extensions.

Listing 1: Using HTM

```
Txfer(&x, &y, amt)
{
    HTMBegin();
    x -= amt;
    y += amt;
    HTMEnd();
}
```

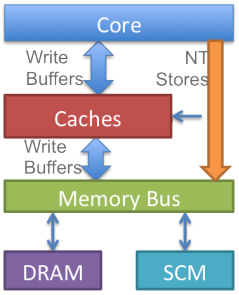

Fig. 1: Stores to SCM

However, these Transactional Memory atomic notions have an interesting concern in view of persistent, byte-addressable SCM. Consider the HTM code fragment shown in Listing 1. A programmer might assume if variables x and y are stored in SCM, then, since the variables are inside an HTM Begin and End block, they are safely updated in persistent storage in case of failure. This is not the case as values may be caught in various cache levels or buffers, evicted out of order, or have not reached SCM home locations at all.

A system with a persistent, or non-volatile, SCM DIMM attached to a memory bus alongside a volatile DRAM DIMM is shown in Figure 1. A write to a variable x might be caught in a number of places from the write buffer to the cache, the cache hierarchy, or in main memory or memory buffer if evicted. Upon completion of an atomic or locked section, the new value of x should be visible and accessible to other threads needing the new value.

However, if the value is located in SCM and a power failure occurs, the result of a completed computation in an atomic block may not be committed to persistent SCM. Even if a value is written to a memory location using a cache-line flush, *clflush*, to force the value out of the cache into main memory, it still may be buffered in the lower buffer of Figure 1 and not stored persistently. Fortunately a new instruction *pcommit* solves this by making all globally-visible memory stores persist once retired. Therefore, programmers might still have to revert to a variety of potentially expensive methods to ensure persistence, such as logging and locking.

II. Transactional Storage Class Memory

TSCM or Transactional Storage Class Memory extends both Hardware Transactional Memory (HTM) and Software Transactional Memory (STM) approaches for concurrency control to additionally support atomic persistence of variables in a managed transaction to SCM.

In Hardware Transactional Memory, the hardware implementation holds writes in a transaction block and doesn't make them visible until successful completion. If the transaction aborts, x and y must retain their original values, and if it commits, the values must be persisted in SCM. Simply flushing x and y to SCM and performing a *pcommit* surrounded by *sfence*s on a transaction commit is not sufficient. If a power failure occurs before y is committed and after x is committed, then the underlying SCM can be inconsistent. Therefore, as x and y are written with their new values during a hardware transaction, TSCM writes to a log area in SCM. These log values are also held in the transaction and not pushed to SCM, as streaming to memory inside a transaction will cause the transaction to abort.

TSCM performs transaction commit in two phases. First, the HTMEnd flushes the log values to the SCM log area, along with the current system clock for log transaction serialization, and performs a persistent commit. Next the values to x and y can be safely written back to their home SCM locations either directly or deferred. Only after the values of x and y are copied and safely committed to their home locations are their logs be removed.

For Software based TSCM we modify Transactional Locking, TL2 [1], to have persistence consistency on transaction completion. In Transactional Locking, locks for write sets may be acquired across words, objects, or configurable stripes. TL has two different modes, a *Lazy or Commit* mode and an *Eager or Encounter* mode. Read validation is performed on counters for the locks and incremented on completed writes. Lazy mode has the benefit of not holding as many locks which can increase performance but may cause more rollbacks.

In the *Commit* mode, writes are committed lazily at the end of the transaction using a linked list write set. Reads first check the write set to get the latest value and a Bloom Filter is used to speed access checking. A read set is also maintained for version checking. Locks are acquired during the commit phase of the transaction, with the transaction aborting if the lock cannot be acquired or the read set fails validation. In TSCM we modify the lazy commit operation to support atomic persistence to SCM, we use the write set as a redo-log. When writes are added to the write set, they are written in a write through manner using streaming stores. On commit, after validating the read set, the write set (or redo-log) is safely committed to SCM. After completion, the write set is write-through copied to the home locations followed by a final persistent memory commit. This implementation of TSCM also has the benefit of asynchronously log updates to SCM.

In *Encounter* mode, the same read and write sets are maintained except that the locks to the writes are acquired as they are encountered in the transaction. TSCM uses the write set to operate like an Undo-Log and the values are safely and synchronously copied and persisted. After the variable is written safely to the log, TSCM persists it to the home location. A pcommit is not needed as it will be committed before the next log update or at the transaction commit. When a transaction is committed successfully, the log is removed.

III. Evaluation

Our TSCM implementation extends RTM TSX locks in hardware mode and STM TL2 for software mode. Testing was performed on a dual socket machine equipped with Intel(R) Xeon(R) CPU E5-2699 v3 processors, for a total of 72 hardware threads. Stanford Transactional Applications for Multi-Processing, or STAMP, is a benchmark suite comprised of eight applications with multiple data sets.

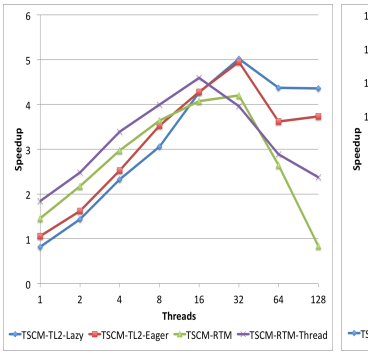

Fig. 2: SSCA

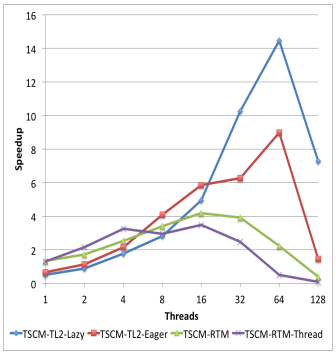

Fig. 3: Vacation Low

The SSCA test is a Scalable Synthetic Compact Applications test that operates on a large graph structure. The transactions are short and write sets small. Figure 2 shows the performance of TSCM on SSCA for various types and number of threads. RTM-Thread uses a background thread to persist log entries. Under high thread contention, TSCM-RTM started to abort and transactions started to serialize, but TSCM-RTM-Thread had the highest performance until about 16 thread workers were active. The vacation benchmark emulates a reservation system in an OLTS with medium length read and write sets. Vacation low shown in Figure 3 had similar performance to SSCA but the trends start much earlier since the vacation transactions are longer and have more writes.

IV. Summary

This paper presents several approaches for combining well-known Transactional Memory techniques to support the durability of transactions on future SCM devices for high performance parallel applications. The techniques were implemented in a library called TSCM, Transactional Storage Class Memory. We found that contention management matters greatly when persisting to SCM; in lighter contention TSCM-RTM performs better than durable software methods.

V. Acknowledgement

I would like to thank Dr. John Mellor-Crummey for inspiring this work during his course at Rice University. Supported in part by Intel Corp SSG grant and NSF grant CCF-1439075.

References

[1] Dice, D., Shalev, O., and Shavit, N. Transactional locking ii. In *Distributed Computing*. Springer, 2006, pp. 194–208.

[2] Herlihy, M., and Moss, J. E. B. *Transactional memory: Architectural support for lock-free data structures*, vol. 21. ACM, 1993.

[3] Rajwar, R., and Goodman, J. R. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Micro'01*, pp. 294–305.