PODC '05 poster abstract

Using LL/SC to Simplify Word-based Software Transactional Memory*

Virendra J. Marathe and Michael L. Scott

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{vmarathe, scott}@cs.rochester.edu

February 2005

Abstract

While *compare-and-swap* (CAS) and *load-linked / store-conditional* (LL/SC) are equally powerful in principle, there are circumstances in which one or the other is significantly easier to use. We highlight one example in this paper. Specifically, we use LL/SC to significantly simplify the "stealing" and "merging" mechanism used to ensure correct nonblocking behavior in Harris and Fraser's word-based software transactional memory system (WSTM). Our simplification exploits the observation that LL/SC, as conventionally implemented, provides a natural atomic implementation of a restricted form of *k-compare-single-swap*.

1 Introduction

An atomic k-compare-single-swap (KCSS) [3, 5] atomically verifies k memory locations and updates one of them (call it t). KCSS is useful for nonblocking implementations of concurrent data structures whose atomic updates require consistent *snapshots*. If the locations involved in the snapshot are always updated in an appropriate order, then ideal LL/SC provides a natural implementation of a restricted form of KCSS: t is load-linked, the remaining locations are read and, if the values satisfy some appropriate predicate, t is updated using store-conditional (SC). The restriction is that whenever the k locations satisfy the predicate, the application must refrain from modifying locations other than t until t itself has been updated.

*This work was supported in part by NSF grants numbers EIA-0080124, CCR-0204344, and CNS-0411127, and by financial and equipment grants from Sun Microsystems Laboratories.

Real implementations of LL/SC impose additional restrictions: most specify that SC can fail spuriously under certain circumstances (e.g. hardware interrupts), in which case software must retry the atomic sequence. More significantly, some allow SC to fail *deterministically* if the instructions between the LL and the SC attempt to access a location that maps to the same cache set as t.

While deterministic failure limits the generality of the technique, LL/SC-based restricted KCSS can still be highly useful if we can guarantee that t lies in a different cache set from the $k-1$ other locations. In particular, LL/SC-based 2CSS can be used to good effect in the word-based software transaction memory system (WSTM) of Harris and Fraser [1, 2]. We describe the original WSTM in Section 2, drawing attention to a potentially significant scalability problem that can arise when contention is high. In Section 3 we present a modification to WSTM that significantly simplifies the design and eliminates the scalability issue, at the expense of one additional atomic instruction on the low-contention critical path. Empirical evaluation of our modification is in progress.

2 Word-based STM

Software Transactional Memory refers to a family of general purpose constructions that can be used to mechanically transform correct sequential code into nonblocking concurrent code. STM systems generally attempt to maximize concurrency by allowing threads to access disjoint sets of *blocks* concurrently. The WSTM of Harris and Fraser [1, 2] makes each memory word a separate block. The API for WSTM

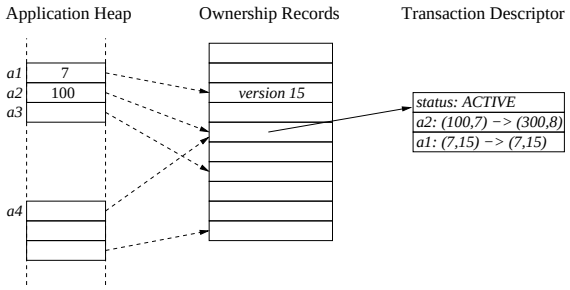

Figure 1: WSTM Heap Structure

has six main entry points: STMStart( ) begins a new transaction. STMRead(addr a) and STMWrite(addr a, stm_word w) are used to read and write shared memory words. STMCommit( ) and STMAbort( ) are used to finalize an ACTIVE transaction. STMValidate( ) verifies that the transaction is still able to commit, which implies that all read locations are mutually consistent.

Figure 2 illustrates the design of WSTM. The *Application Heap* is the shared memory region that holds the actual data. The structure in the middle of the figure is a hash table of *Ownership Records* (orecs). Each orec stores ownership (permission-to-modify) information for all memory words that hash to its index in the table. Each *unacquired* orec stores a version number. A transaction has to *acquire* an orec before modifying any corresponding memory words. Acquiring consists of atomically replacing the orec version number with a pointer to the acquiring transaction's descriptor.

A transaction descriptor contains a list of *transaction entries*, one for each shared word in the application heap accessed by the transaction. Each transaction entry in turn has five fields: the address of the shared word, the original contents of that word, the version number of the corresponding orec, the new contents of the memory word (to be written back at commit time), and the new version number (to be stored in the orec).

A transaction may be ACTIVE, ABORTED, or COMMITTED. An STMRead or STMWrite creates a transaction entry (corresponding to the accessed memory location) if one does not already exist in the transaction descriptor. To maintain consistency, a transaction descriptor must either contain at most one entry corresponding to an orec, or all the entries corresponding to an orec must have the same old and new version numbers. The STMCommit operation attempts to acquire all orecs named in the transaction descriptor. If successful it uses an atomic primitive (the linearization point of the transaction) to switch to COMMITTED state. It then updates the shared heap and *releases* all acquired orecs by swapping in their new version numbers.

An STMRead or STMWrite to a previously unaccessed location inspects the corresponding orec. It the orec points to an ACTIVE transaction, the contender is immediately aborted (this uniform aggressiveness raises the possibility of livelock, making WSTM obstruction-free [4]). The current transaction creates a transaction entry using the appropriate orec version number (old if ABORTED, new if COMMITTED) found in the contender's descriptor.

An STMCommit that discovers a conflict also aborts the contender if it is still ACTIVE. It then *merges* transaction entries (corresponding to the orec under conflict) from the contender's descriptor into its own. Merging allows the current transaction, once it finalizes, to appropriately update any locations for which the contender was responsible, even if the contender is preempted or otherwise inactive. After merging, the current transaction *steals* the orec from its contender by using an atomic primitive to flip the pointer over to its own transaction descriptor.

Use of stealing leads to the problem of stale updates, where a transaction that is a victim of stealing may update words in the heap after the stealer has already done so. A victim realizes this potential problem when it tries to release the stolen orec and its CAS or SC fails. The victim then chases the orec pointer to its stealer's descriptor and *redoes* all updates made by the stealer for the stolen orec. No orec is released until it is guaranteed that the orec is not referenced by any other transaction. This is enforced by the use of a reference count for each orec. An orec is released by a transaction only when the orec reference count goes down to zero. Atomic update to the orec and its reference count requires a double-wide CAS or LL/SC.

Bounded Memory Blow-up

WSTM uses stealing to ensure nonblocking semantics. Stealing entails merging, which in turn leads to potentially long *merge chains* of transaction entries

due to *false sharing*. Let the ratio of the application heap size to the orec hash table size be $M:1$. If hashing is uniform, each orec covers approximately M different shared memory words. In the worst-case scenario, for each memory location that a transaction may access, it may end up possessing $M-1$ extra transaction entries. If a transaction needs to acquire N orecs, it may end up possessing $N \times M$ transaction entries in the process. Memory blow-up may be significant if M is large. Responsibility for extra memory words also increases the worst-case overhead of write back by a factor of M, and introduces the overhead of redos, which may cause severe interconnect contention as cache lines bounce among processors. Although the worst case scenario may rarely occur, its likelihood increases with increasing contention.

3 An Alternative Stealing Approach

WSTM's stealing mechanism leads to the bounded memory blow-up problem and potentially slower transactions. It also introduces significant complexity, and requires a double-wide atomic primitive (not currently available on 64-bit processors) to update version number / pointer pairs. We propose an alternative mechanism that uses *helping* during stealing instead of *merge-redo*. On detecting a conflict, a potential stealer transaction first scans through its victim's descriptor looking for the transaction entries corresponding to the orec under conflict. For each such entry, the stealer updates the corresponding shared memory location using LL/SC to implement a restricted 2-compare-single-swap: the stealer LLs the heap location, verifies that the orec is the same as the one in the transaction entry, and then stores the right value with SC. After the stealer has scanned through its victim's descriptor, it steals the orec under conflict as before. The victim will continue with its release phase normally (without updating memory words corresponding to the stolen orec) even when it sees that some of its orecs have been stolen.

The intuition behind our approach is as follows: If the orec changes after an LL, it is guaranteed that some other stealer has successfully stolen the orec after making correct updates to the heap. The transaction will have to chase the new stealer to resolve the new conflict with its new contender. If the orec is still valid, but the SC fails non-spuriously, it is guaranteed that some other potential stealer has made a correct update to the target memory location. Spurious failures are handled by a retry loop. Stale updates are avoided by verifying the orec contents in between the LL and SC. With our approach the bounded memory blow-up problem is eliminated since no merging of transaction entries happens. The commit operation for a transaction is also simplified considerably. Finally, no reference counts or double-wide atomic operations are required. To avoid deterministic SC failures, we need only ensure that a heap location and its orec never map to the same set in the cache. This is easily achieved, for all reasonable cache line and page sizes, and for both virtually and physically indexed caches, by selecting an appropriate hash function.

Our modification has a downside: a transaction updating N memory words requires $N+2M+1$ LL/SC operations (where M is the number of orecs acquired by the transaction), versus $2M+1$ CASes in the original WSTM. Experimental evaluation of this tradeoff is currently in progress.

References

[1] K. Fraser and T. Harris. Concurrent Programming without Locks. *Submitted for publication*, 2004.

[2] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proceedings of 18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, October 2003.

[3] T. L. Harris, K. Fraser, and I. A. Pratt. A Practical Multi-word Compare-and-Swap Operation. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 265–279. Springer-Verlag, 2002.

[4] M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of 23rd International Conference on Distributed Computing Systems*, pages 522–529, May 2003.

[5] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 314–323, June 2003.