Unmanaged Multiversion STM *

Li Lu Michael L. Scott
University of Rochester
{llu,scott}@cs.rochester.edu

Abstract

Multiversion STM systems allow a transaction to read old values of a recently updated object, after which the transaction may serialize *before* transactions that committed earlier in physical time. This ability to "commit in the past" is particularly appealing for long-running read-only transactions, which may otherwise starve in many STM systems, because short-running peers modify data out from under them before they have a chance to finish.

Most previous work on multiversion STM assumed an object-oriented, garbage-collected language. In contrast, we present a multiversion STM system designed for unmanaged languages, in which ownership information is typically maintained on a "per-slice" (rather than a "per-object") basis, and in which automatic garbage collection may not be available. We have implemented this UMV system as an extension to the TL2-like LLT back end of the RSTM suite. Experiments with microbenchmarks confirm that UMV can dramatically reduce the abort rate of long read-only transactions. Overall impacts range from 2× slowdown to 5× speedup, depending on the offered workload.

1. Introduction

Long-running transactions, even if read-only, pose a challenge for many STM systems: if transaction T reads location x early in its execution, it will typically be able to commit only if no other thread commits a change to x while T is still active. Since readers are "invisible" in most STM systems, writers cannot defer to them, and a long-running reader may starve. The typical solution is to give up after a certain number of retries and re-run the long reader under the protection of a global lock, excluding all other transactions and making the reader's completion inevitable (irrevocable).

A potentially attractive alternative, explored by several groups, is to keep old versions of objects, and allow long-running readers to "commit in the past." Suppose transaction R reads x, transaction W subsequently commits changes to x and y, and then R attempts to read y. Because the current value of y was never valid at the same time as R's previously read value of x, R cannot proceed, nor can it switch to the newer value of x, since it may have performed arbitrary computations with the old value. If, however, the older version of y is still available, R can safely use that instead. Assuming that the STM system is otherwise correct, R's behavior will be the same as it would have been if it completed before transaction W.

Multiversioning is commonplace in database systems. In the STM context, it was pioneered by Riegel et al. in their SI-STM [15] and LSA [14] systems, and, concurrently, by Cachopo et al. in their JVSTM [4, 5]. SI-STM and LSA maintain a fixed number of old versions of any given object. JVSTM, by contrast, maintains all old versions that might potentially be needed by some still-running transaction. Specifically, if the oldest-running transaction began at time t, JVSTM will keep the newest version that is older than t, plus all versions newer than that.

In all three systems, no-longer-wanted versions are manually deleted by breaking the last pointer to them, after which the standard garbage collector will eventually reclaim them. More recently, Perelman et al. [12] demonstrated, in their SMV system, how to eliminate manual deletion: they distinguish between *hard* and *weak* references to an object version v, and arrange for the last hard reference to become unreachable once no running transaction has a start time earlier than that of the transaction that overwrote v.

Several additional systems [1, 3, 9–11] allow not only readers but also writers to commit in the past. Unfortunately, because such systems require visible readers and complex dependence tracking, they can be expected to have significantly higher constant overheads. We do not consider them further here.

SI-STM, LSA, JVSTM, and SMV were all implemented in Java. While only SMV really leverages automatic garbage collection, all four are "object-based": their metadata, including lists of old versions, are kept in object headers. One might naturally wonder whether this organization is a coincidence or a necessity: can we create an efficient, multiversion STM system suitable for unmanaged languages like C and C++, in which data need not be organized as objects, and in which unwanted data must always be manually reclaimed?

Our UMV (Unmanaged MultiVersioning) system answers this question in the affirmative. As described more fully in Sec. 2, UMV is a "slice-based" STM system in the style of TL2 [7] and TinySTM [8]. It maintains an array of *ownership records* (Orecs), in which metadata are shared by all locations that hash to a given slot in the array. Timestamps on Orecs are used to identify conflicts among writers.

* This work was supported in part by the National Science Foundation under grants CCR-0963759, CCF-1116055, and CNS-1116109.

In contrast to most systems, however, UMV attaches chains of old data versions to each Orec, so readers can work in the past. When memory runs low, any thread can peruse the Orec table and reclaim any versions that predate the oldest still-active transaction.

We have implemented UMV as a new back end in the RSTM suite [13]. It is based on the existing LLT back end, which resembles TL2. It is not privatization safe, though this could certainly be added. Experiments with microbenchmarks (Sec. 3) suggest that UMV successfully eliminates the starvation problem for long-running readers, yielding dramatically higher throughput than single-version systems for workloads that depend on such transactions. The penalty is higher constant overhead—a reduction in throughput for read-write transactions of up to 2× in our (as yet unoptimized) implementation. Additional issues, including how to identify transactions that ought to run in the past and how to tune garbage collection, are discussed in Sec. 4.

2. Multi-versioning TL2 Design

In this section we introduce the algorithm design of UMV. Generally, on writer transaction commits, UMV will back up all changed values to a history list. When a read-only transaction discovers that a concurrent write may have changed the value in a location it is reading, it will search for and then return an appropriate old value from a history list. In this way, read-only transactions will never abort. In order to reclaim unused history version data, we include a garbage collection mechanism.

2.1 Algorithm design

As suggested in Sec. 1, UMV can be described as an extension to timestamp-based STM systems like TL2. As in "stripe-based" TL2 [7], each entry in a global *Orec table* stores metadata for locations that hash to that entry. Each entry (Orec) contains a lock bit, a version number, and, for multiversioning, a history list. The history list stores all historical values that may be used for a stripe. Each node n in the list has three fields: a memory location `addr`, a `value` formerly contained in `addr`, and the global timestamp (version number) `ts` when this value was overwritten.

Nodes are kept in time-sorted order, with the newest entries at the head. For a given list node n, if n' is the next-older entry in the same list such that $n.\texttt{addr} = n'.\texttt{addr}$, we know that location $n.\texttt{addr}$ held value $n.\texttt{value}$ from time $n'.\texttt{ts}$ up to but not including time $n.\texttt{ts}$.

UMV differs from TL2 in its code for writer transaction commits, read-only transaction reads, and garbage collection. On writer transaction commits, UMV must back up all changed values into history lists. On a shared-memory read in a read-only transaction T, if the relevant Orec has a version number newer than T's start time, instead of aborting, T will traverse the history list and return an historical value.

```
1  W_Begin_Transaction():
2     tx.start_time = timestamp
3
4  W_Read(address l):
5     v = tx.write_set.find(l)
6     if (found) return v
7     tx.read_set.add(l)
8     o = get_orec(l)
9     ts1 = o.time_stamp
10    v = *l
11    ts2 = o.time_stamp
12    if ((ts2 <= tx.start_time)
13       && (ts1 == ts2)) return v
14    else abort();
15
16 W_Write(address l, value v):
17    tx.write_set.add(l, v)
18
19 W_Commit():
20    foreach write_record w in tx.write_set
21       o = get_orec(w.addr)
22       bool success = o.lock.acquire()
23       if (!success) abort();
24       if (!tx.lock_set.find(o))
25          tx.lock_set.add(o);
26    end_time = 1 + fai(timestamp)
27    Validate()
28    foreach write_record w in tx.write_set
29       o = get_orec(w.addr)
30       n = new history_node(end_time,
31             w.addr, *w.addr)
32       o.history_list.add_at_head(n)
33       *w.addr = w.value
34    foreach orec o in tx.lock_set
35       o.unlock_and_set_ts(end_time)
36    if (need_gc()) gc()
37
38 Validate():
39    foreach read_record r in tx.read_set
40       o = get_orec(r.addr)
41       if (o.ts > tx.start_time ||
42             (o.is_locked()
43                && lock.owner != tx.id))
44          abort()
```

Figure 1. UMV pseudocode for a writer transaction `tx`.

```
R_Begin_Transaction():
   tx_times[tid] = tx.start_time =
       timestamp

R_Read(address l):
 begin:
   o = get_orec(l)
   if (o.is_locked()) goto begin
   ts1 = o.time_stamp
   v = *l
   if (o.is_locked()) goto begin
   ts2 = o.time_stamp
   if (ts1 != ts2) goto begin
   if (ts2 <= tx.start_time)
      return v
   n = o.history_list.head
   prev = NULL
   while (n != NULL)
      if (n.ts <= tx.start_time)
         break
      if (n.addr == l)
         prev = n
      n = n.next
   if (prev != NULL)
      return prev.value
   else
      return v

R_Commit():
   tx_times[tid] = maxint
```

Figure 2. UMV pseudocode for a read-only transaction `tx`.

On the commit of a writer transaction W, UMV will perform the operations shown in Fig. 1. These are the same as in TL2, with the addition of history list insertions and garbage collection:

1. For each entry in W's write set, try to lock the corresponding Orec. If unable to do so, abort.
2. Increase the global timestamp by one, and remember the result as W's end time.
3. Validate W's read set. If unsuccessful, abort.
4. For each entry in W's write set, push a node containing the old value onto the history list of the corresponding Orec. Write the new value back to memory.
5. For each entry in W's write set, release the lock on the corresponding Orec.
6. If significant time has passed since the last check, sum up per-thread counts of active history nodes. If the value is too high, call the garbage collector.

On a shared-memory read r of location l, in a read-only transaction R, UMV will perform the operations shown in Fig. 2:

1. As in TL2, find the Orec o corresponding to l, and check its timestamp to see if l may have been modified recently. If not, return the value of a normal memory read.
2. Otherwise, peruse o's history list, looking for the last (oldest) node whose address field is l and whose overwrite time is greater (newer) than R's start time.
3. If such a node is found, return its value; otherwise, return the value of the normal memory read.

2.2 Garbage Collection

To avoid unbounded memory growth, history lists must periodically be pruned. This pruning is facilitated by the following observation: in the code of Fig. 2, no transaction T will travel down a history list beyond the first node whose time is less than or equal to T's start time. If t_{min} is the minimum start time across all active transactions, we can be sure that anything with time less than or equal to t_{min} will never be needed again, and can safely be reclaimed.

To enable calculation of t_{min}, we maintain a global array `tx_times`, indexed by thread id. Each reader transaction begins by moving the global timestamp into its entry in `tx_times`. It ends by setting its entry to infinity. The minimum value found in a (possibly non-atomic) scan of the array is then guaranteed to be less than or equal to the start time of the oldest active reader.

To determine when garbage collection is needed, we maintain another global array, which records the number of history nodes allocated by each thread. In addition, a global counter stores the number of nodes that have been released by the garbage collector. After a predefined (but tunable) number of its own writer transaction commits, a thread will sum up the global array of allocation counts, subtract off the global release count, and invoke the garbage collector if the difference exceeds another predefined (but tunable) threshold. While other GC strategies are possible, this has the advantage of requiring no additional dedicated resources: the work of collection is spread among the threads, and can occur in parallel with continued execution in other threads. Additional discussion of garbage collection appears in Sec. 4.2.

3. Performance Evaluation

In this section we present preliminary data on the baseline cost of multiversioning, and the impact on fairness/starvation.

3.1 Baseline Overhead

Fig. 3 assesses the overhead of creating, garbage collecting, and optionally consulting history lists. The experimental workload is a simple hash table microbenchmark consisting of lookup, insert, and delete operations (all small), in an 80:10:10 mix. (Half the inserts find the element already present, and half the deletes find it absent, so 90% of the operations are actually read-only.) The table has 4096 buckets, and is populated with keys drawn from the range 0..4095.

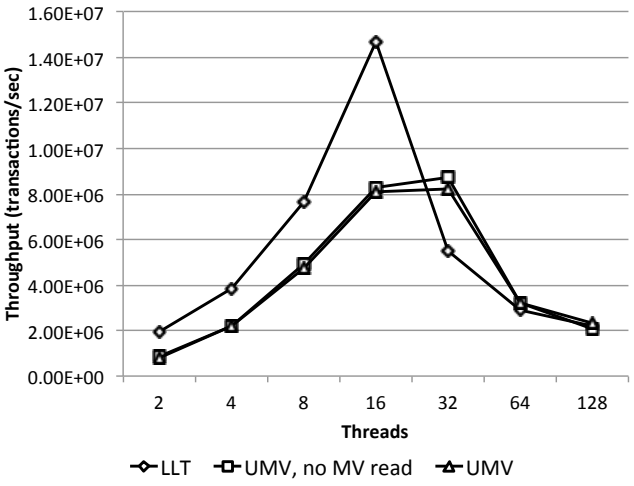

Figure 3. Throughput of simple hash table benchmark on the Niagara 2. History lists are created in both UMV variants, but consulted in only one.

After initial warm-up (not counted in the timings), the table is typically half full.

The hardware is a two-processor Oracle Niagara 2. Each processor has 8 in-order, dual-issue cores running at 1.2 GHz, and 8 hardware threads per core (4 threads per pipeline). Each core has 24KB of L1 D-cache; the cores of a given processor share 4MB of on-chip L2 cache.

Compared to LLT, UMV reduces overall throughput by roughly 35% - 40%. Because the transactions are all small, there is no real benefit to be gained from history lists, and in fact readers seldom encounter an orec with a too-new timestamp (hence the near-identical performance with and without consultation of history lists, marked as "UMV" and "UMV, no MV read" respectively in the figure). While the LLT code is mature and well optimized, the UMV extensions are not; we have hopes that the baseline overhead will decrease with further work.

We do observe that with 32 threads, UMV outperforms LLT, which is counterintuitive. After investigation, it appears that the extra work required to update history lists in writers has the unexpected consequence (for high thread counts) of reducing contention with readers, resulting in a more-than-compensating increase in overall throughput.

3.2 Throughput

To assess the potential benefit of multiversioning, we modify our hash table microbenchmark to include long-running "sum" operations, which traverse the entire table and add up all its elements. Unlike lookup operations, which are small and fast, sum operations take long enough that they almost always conflict with concurrent writers, and will starve unless something special is done.

Fig. 4 presents results for three different workload mixes: a write-heavy test (leftmost graph) with 1:19:80 ratios of sum, lookup and update (insert or delete) operations; a "balanced" test (middle graph) with ratios of 1:49:50; and a read-heavy test (rightmost graph) with ratios of 1:79:20. (Again, because half the updaters don't actually write, the ratios of read-only transactions to read-write transactions are 60:40, 75:25 and 90:10, respectively).

For each workload mix we compare the throughput (transactions/sec) of LLT, UMV, and two variants of the simpler NOrec algorithm [6]. Because NOrec serializes transaction write-back using a global lock, it supports a trivial implementation of inevitability (irrevocability). In the "NOrec inevitable" experiments we use inevitable mode to run the sum transactions. We also test a (non-general) extension of LLT (labeled "LLT inevitable") in which the checker thread acquires a global lock, which other threads read, forcing them to abort, if active, and wait to retry.

LLT, UMV, and (basic) NOrec all scale up to 64 threads. In the read-heavy workload, UMV even obtains a tiny bit of additional throughput at 128 threads. Significantly, the serialization of long-running inevitable transactions has a terrible impact on performance. Though sum operations are only 1% of the dynamic transaction mix, they take many times as long to execute, with the result that throughput scales almost not at all. (As we shall see in the following subsection, fairness is a different, more complicated matter.)

In all three job mixes, given the fact that every thread attempts to perform occasional long-running reader transactions, multiversioning leads to dramatically higher overall throughput. While UMV imposes history-maintenance costs on update transactions, it ensures that the long-running readers never abort.

3.3 Fairness

As noted in the previous subsection, inevitability can be used to prevent starvation in long-running readers, but at significant cost in scalability. To better assess the impact on starvation and, more generally, fairness, we consider a revised microbenchmark in which all long-running transactions are performed by a single thread, rather than appearing within the mix of transactions in every thread. Specifically, we create a workload in which all threads but one perform a mix of small lookup and update transactions, while an additional "checker" thread repeatedly scans the table for "anomalies" (in our code, nodes with a value greater than some constant c). Normal threads execute a tight loop of lookup and update operations in an 80:20 ratio (90% read-only). The checker thread attempts a scan once every millisecond.

For all algorithms, we collect two measures of performance: the completion rate of check transactions and overall throughput.

In the checker thread (Fig. 5), the completion rate for LLT is close to zero: the previous scan has almost never completed successfully when the next millisecond rolls around, and the checker starves. The two algorithms with inevitability have finish rates above 90% at all thread counts. For

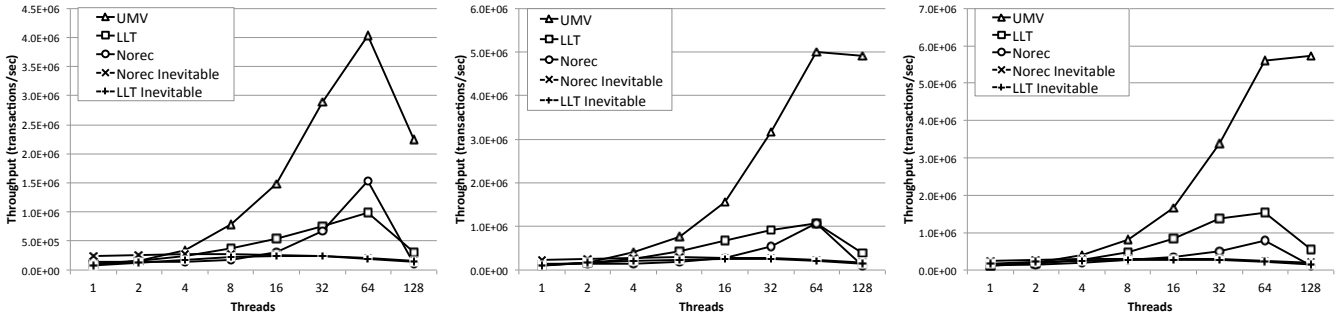

Figure 4. Throughput with 1% long-running reader transactions on a 16-core 128-hardware-thread Niagara 2 machine. From left to right the percentage of update operations is 80%, 50%, and 20%, respectively (40%, 25% and 10% writer transactions).

UMV, the finish rate drops precipitously from 32 to 64 threads. In this case check transactions never abort, so the source of the problem is different than in LLT. Given the architecture of the machine, we conjecture that the problem is a lack of hardware resources. Specifically, the two processors provide a total of 32 execution pipelines. With more than 32 active threads, the checker does not get exclusive use of a pipeline, and cannot complete in 1ms.

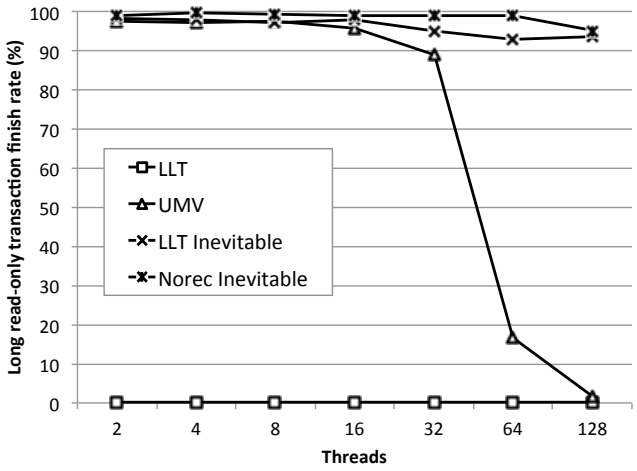

Figure 5. Finish rate for long read-only transactions.

In Fig. 6, UMV, LLT, and NOrec all scale well to 16 threads. Beyond this point, LLT and NOrec suffer unacceptable contention, and performance declines. It declines sooner than in the experiments of the previous subsection because all the time is being spent in small transactions; the worker threads have no long-running sum operations to slow them down. For similar reasons, UMV continues to scale to 32 threads: the extra overhead of history list maintenance delays the point at which contention becomes unacceptable. We omit results for plain LLT, and for UMV beyond 32 threads, where the checker is unable to keep up.

Given these results, we consider multiversioning an attractive alternative to inevitability. So long as long-running reader threads aren't starved for computational resources

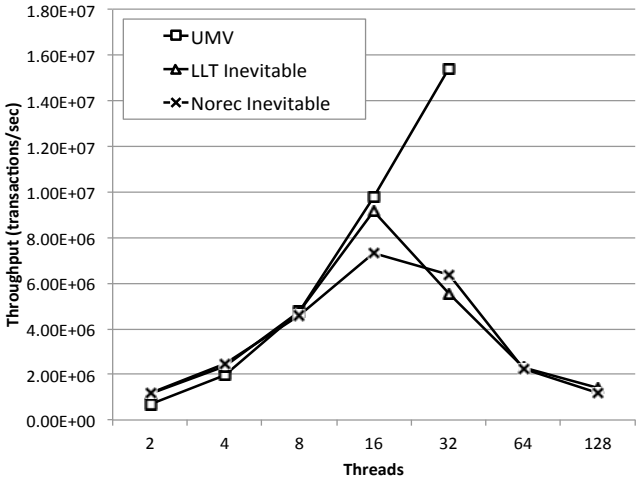

Figure 6. Throughput in the presence of long read-only transactions. In the interest of fairness, we report throughput data only when long read-only operations have a finish rate greater than 85%.

(e.g., pipeline slots), multiversioning allows them to complete without aborting, and to co-exist with short transactions that continue to scale to the limits otherwise imposed by the STM runtime and hardware coherence fabric.

4. Discussion

In this section, we consider two design issues for multiversioning STM systems: the identification of long-running reader transactions and the tuning of garbage collection. In each case we describe our current solution, outline alternatives, and suggest directions for future work.

4.1 Identifying Long-running Readers

In many STM systems, transactions that may execute in more than one "mode" must be compiled into multiple *clones*, with different read and write barriers (instrumentation) in each. Recent releases of RSTM, however, incorporate a dynamic reconfiguration mechanism that accesses barrier code through an indirection table that is subject to

```
Begin_Transaction()
   curr = TM_Read(list→head)
   while (curr != NULL)
      if (TM_Read(curr→id) == thread_id)
         break
      curr = TM_READ(curr→next)
   if (curr != NULL)
      TM_Write(curr→val, curr→val + 1)
Commit()
```

Figure 7. Static annotation of read-only transactions may not work. Here whether the transaction is read-only can only be decided at run time.

```
Begin_Transaction()
   for (int i = 0; i < len; i++)
      TM_Read(list[i])
      if (i == THRESHOLD)
         TM_Use_MV()
Commit()
```

Figure 8. A transaction that chooses to start multiversioning dynamically. The `TM_Use_MV` function will switch to multiversion mode iff the transactions is still read-only.

dynamic update [16]. In LLT, for example, the initial read barrier ignores the (empty) write log. The first execution of a write barrier, if any, replaces both the read and write barriers, so that the former checks the write log (ensuring that transactions see their own writes) and the latter elides the change-over code.

For UMV, the RSTM adaptation mechanism allows us to use the same code path for read-only and read-write transactions: all that differs is the indirection table. In fact, we can switch between read-only and read-write instrumentation not only at run time, but even in the middle of a transaction. This capability may be useful for transactions in which static annotation may not maximize system performance. In Fig. 7, for example, if we assume that the desired node is found about half of the time, the other half of the time the transaction will be read-only. If we choose read-write mode statically, we are likely to suffer unnecessary aborts in transactions that never really write. If we choose multiversion mode, we will need to abort and restart whenever a transaction that needs to write has read an historic (no longer current) value. (Of course, if we ran in read-write mode from the outset, the transaction would have aborted when it first encountered the need for an historic read, since read-write transactions are not permitted to commit in the past.)

It is worth noting that even for read-only transactions, multiversioning may not always be beneficial. In a small read-only transaction, if a to-be-read location has changed since the beginning of the transaction, it may be cheaper to abort and restart than to traverse history lists on all future reads. In short, multiversioning makes sense only for transactions that are both read-only and long-running.

In the current version of our code, we provide an executable `TM_Use_MV` operation that attempts to switch the current transaction into multiversion mode. If there has already been a transactional write, the operation is a no-op. Using `TM_Use_MV`, programmers can choose to keep the conventional TL2-like algorithm for most transactions, and enable multiversioned reads only when desired (presumably for long read-only transactions).

If transactions vary in size, the programmer may choose to enter multiversion mode only after a read-only transaction reaches a certain size, as suggested in Fig. 8. Here readers will abort only when they are short, and will pay the overhead of history list searches only when long. Unfortunately, to support the ability of readers to commit in the past (whether they use that ability or not), writers must create history list nodes in all cases. As shown in Fig. 3, this has nontrivial cost.

The principal disadvantage of `TM_use_MV` is the need to invoke it explicitly. In the interest of portability and programming simplicity, it may instead be desirable to automate the selection of multiversion mode. There are many possible ways to do so—using static analysis or profiling, for example. Though we do not employ it in the experiments of Section 3, our current code supports an optional on-line strategy: switch to multiversion mode if (a) we attempt to read a location that has changed since the beginning of the transaction; (b) we have performed at least k transactional reads, for some fixed constant k; and (c) the write log is currently empty. This strategy requires that we maintain a read count (an overhead not normally present in the TL2 read barrier), but it introduces no new branches in the common case: TL2 must already check for out-of-date timestamps.

As observed by Perelman et al. [12], the *timestamp extension* of Riegel et al. [8] is orthogonal to multiversioning. Instead of (1) aborting or (2) switching to multiversion mode, there is an additional alternative when we discover that an about-to-be-read location has changed since the beginning of the transaction: we can (3) re-read all previous locations and, if none has changed since the beginning of the transaction, update our local timestamp and pretend we started now. This strategy requires that we maintain a read log, so we know which locations to re-read. On the plus side, the on-line strategy for switching to multiversion mode becomes essentially free: task (b) in the previous paragraph can be performed by checking the current length of the read log.

4.2 Garbage Collection

Garbage collection is essential for multiversion STM. As described in Sec. 2.2, history nodes in UMV can safely be reclaimed when they represent overwrites that occurred before the beginning of the oldest running read-only transaction.

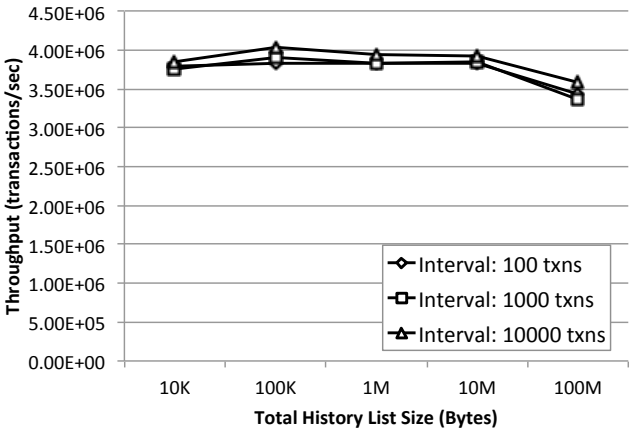

Figure 9. Test result for different combinations of GC interval and history lists memory consumption.

The principal remaining question is when to look for such reclaimable nodes.

Our current strategy, again described in Sec. 2.2, employs two tunable parameters: `GC_INTERVAL` and `GC_THRESHOLD`. The run-time system will assess the amount of memory currently in use for history lists every `GC_INTERVAL` update transaction commits, and will actually peruse the lists if their total size is greater than `GC_THRESHOLD`. Basically, `GC_INTERVAL` controls the frequency at which the GC function is launched, while `GC_THRESHOLD` controls the upper bound on memory usage. If `GC_INTERVAL` is too small, the system may incur unnecessary overhead for frequent perusal of the global array of allocation counts (likely incurring a cache miss on each element); if `GC_INTERVAL` is too large, peak memory usage may be excessive. Likewise, if `GC_THRESHOLD` is too small, the system may incur unnecessary overhead for frequent perusal of history lists; if `GC_THRESHOLD` is too large, peak memory usage may again be excessive.

As a simple form of sensitivity analysis, we ran UMV on our Niagara 2 machine with various combinations of `GC_INTERVAL` and `GC_THRESHOLD` and measured the throughput of the hash table microbenchmark described in Sec. 3.2. Results appear in Fig. 9. Our principal conclusion is that, at least for this program, the choice of parameters doesn't really matter very much. Throughput appears to peak with less frequent checks of usage (every 10,000 transactions), and with mid-range bounds on the size of the heap. With smaller checking intervals, or with unreasonably large heap sizes (100 MB), performance decreases slightly. (The 8MB on-chip cache of the Niagara 2 is not quite large enough to hold the 10MB data set at the right end of the graph.)

Our current results were obtained with the general-purpose Hoard memory manager [2]. Given that history nodes are all the same size, we prepended a custom local allocator. In addition to reducing allocation overhead, local management allows us to reclaim the entire tail of a history list in constant time. Nodes are returned to Hoard (and made available for global use) only when the local pool becomes unreasonably large.

A potentially interesting subject for future work would be the reclamation of *internal* nodes in history lists. Our current algorithm removes only those nodes that are older than the oldest read-only transaction in the system. Some younger nodes, however, could in principle also be reclaimed. For example, if the only two running transactions have timestamps 20 and 30, while in one history list there are nodes with timestamps 21, 22, 26 and 32, only the nodes with times 21 and 32 have any chance of being used. The others could safely be reclaimed, though it could be difficult in general to figure this out efficiently.

5. Conclusions

We have designed and prototyped a multiversion STM system (UMV) for C and C++. In comparison to the TL2-like system on which it is based, it displays up to 5× improvement in throughput for workloads that depend on long-running readers, and up to 2× slowdown for workloads that do not. With further implementation effort, the overhead of UMV can probably be reduced, but for small transactions it will always be higher than the baseline. Topics for future work include (1) better automatic mechanisms to choose when to enter multiversion mode; (2) a new mechanism to choose (on a global basis), whether writers should maintain history lists; and (3) an efficient mechanism to garbage collect unneeded internal history nodes.

References

[1] U. Aydonat and T. Abdelrahman. Serializability of Transactions in Software Transactional Memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, UT, Feb. 2008.

[2] E. Berger, K. McKinley, R. Blumofe, and P. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Nov. 2000.

[3] A. Bieniusa and T. Fuhrmann. Consistency in Hindsight: A Fully Decentralized STM Algorithm. In *Proc. of the 24th Intl. Parallel and Distributed Processing Symp.*, Atlanta, GA, Apr. 2010.

[4] J. Cachopo and A. Rito-Silva. Versioned Boxes as the Basis for Memory Transactions. In *Proc., Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, Oct. 2005. In conjunction with *OOPSLA '05*.

[5] J. Cachopo and A. Rito-Silva. Versioned Boxes As the Basis for Memory Transactions. *Science of Computer Programming*, 63(2):172-185, Dec. 2006.

[6] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In

Proc. of the 15th ACM Symp. on Principles and Practice of Parallel Programming, Bangalore, India, Jan. 2010.

[7] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, pages 194-208, Stockholm, Sweden, Sept. 2006.

[8] P. Felber, T. Riegel, and C. Fetzer. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming*, pages 237-246, Salt Lake City, UT, Feb. 2008.

[9] I. Keidar and D. Perelman. On Avoiding Spare Aborts in Transactional Memory. In *Proc. of the 21st ACM Symp. on Parallelism in Algorithms and Architectures*, pages 59-68, Calgary, AB, Canada, Aug. 2009.

[10] J. Napper and L. Alvisi. Lock-Free Serializable Transactions. Technical Report TR-05-04, Dept. of Computer Sciences, Univ. of Texas at Austin, Feb. 2005.

[11] D. Perelman, R. Fan, and I. Keidar. On Maintaining Multiple Versions in STM. In *Proc. of the 29th ACM Symp. on Principles of Distributed Computing*, pages 16-25, Zurich, Switzerland, July 2010.

[12] D. Perelman, A. Byshevsky, O. Litmanovich, and I. Keidar. SMV: Selective Multi-Versioning STM. In *Proc. of the 25th Intl. Symp. on Distributed Computing*, Rome, Italy, Sept. 2011.

[13] Reconfigurable Software Transactional Memory Runtime. Project web site. `code.google.com/p/rstm/`.

[14] T. Riegel, P. Felber, and C. Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[15] T. Riegel, C. Fetzer, and P. Felber. Snapshot Isolation for Software Transactional Memory. In *1st ACM SIGPLAN Workshop on Transactional Computing*, Ottawa, ON, Canada, June 2006.

[16] M. F. Spear. Lightweight, Robust Adaptivity for Software Transactional Memory. In *Proc. of the 22nd ACM Symp. on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.