Generic Multiversion STM*

Li Lu and Michael L. Scott

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

Abstract. Multiversion software transactional memory (STM) allows 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 approaches to multiversioning have been designed as an integral part of some larger STM system, and have assumed an object-oriented, garbage-collected language. We describe, instead, how multiversioning may be implemented on top of an almost arbitrary "word-based" STM system. To the best of our knowledge, ours is the first work (for any kind of STM) to combine bounded space consumption with guaranteed wait freedom for read-only transactions (in the form presented here, it may require writers to be blocking). We make no assumptions about data or metadata layout, though we do require that the base system provide a hash function with certain ordering properties. We neither require nor interfere with automatic garbage collection. Privatization safety can be ensured—without compromising wait freedom for readers—either by forcing privatizing writers to wait for all extant readers or by requiring that programmers explicitly identify the data being privatized.

1 Introduction

Transactional memory (TM) raises the level of abstraction for synchronization, allowing programmers to specify what should be made atomic without specifying *how* it should be made atomic. The underlying system then attempts to execute nonconflicting transactions in parallel, typically by means of speculation. Hardware support for TM has begun to reach the market, but software implementations (STM) can be expected to remain important for many years.

In both hardware and software TM, strategies for detecting and recovering from conflicts differ greatly from one implementation to another. Most systems, however,

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

Y. Afek (Ed.): DISC 2013, LNCS 8205, pp. 134–148, 2013.
© Springer-Verlag Berlin Heidelberg 2013

have particular trouble accommodating long-running transactions. When *writer* transactions (those that update shared data) conflict with one another (or appear to conflict due to limitations in the detection mechanism) users will presumably not be surprised by a lack of concurrency: in the general case, conflicting updates must execute one at a time. When writers conflict with *readers*, however (i.e., with transactions that make no changes to shared data), one might in principle hope to do better, since there is a moment in time (the point at which it starts) when a reader could execute to completion without interfering with the writer(s).

The problem, of course, is that changes made by writers after a reader has already started may prevent the reader from completing. Specifically, if transaction R 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 R is still active. Since readers are "invisible" in most STM systems (they refrain from modifying metadata, to avoid exclusive-mode cache misses), writers cannot defer to them, and a long-running reader may starve. To avoid this problem, most systems arrange for a long-running reader to give up after a certain number of retries and re-run under the protection of a global lock, excluding all other transactions and making the reader's completion inevitable.

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 all its work before transaction W, and then took a long time to return.

Multiversioning is commonplace in database systems. In the STM context, it was pioneered by Riegel et al. in their SI-STM [21] and LSA [20] systems, and, concurrently, by Cachopo et al. in their JVSTM [3, 4]. 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, the runtime deletes no-longer-wanted versions explicitly, by breaking the last pointer to them, after which the standard garbage collector will eventually reclaim them. More recently, Perelman et al. demonstrated, in their SMV system [17], how to eliminate explicit 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, 2, 11, 16, 18] 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 meta-data, 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 explicitly reclaimed?

Our GMV (Generic MultiVersioning) system answers this question in the affirmative. It is designed to interoperate with any existing "word-based" (i.e., hash-table–based) STM system that provides certain basic functionality. It is also, to the best of our knowledge, the first mechanism to simultaneously (a) guarantee wait-free progress for all read-only transactions, and (b) bound total space consumption—specifically, to $O(nm)$, where n is the number of threads and m is the space consumed by an equivalent nontransactional, global-lock-based program (this assumes reasonable space consumption in the underlying STM system). Finally, GMV can preserve both privatization safety (for writers) and wait freedom for readers if we are willing either to force privatizing writers to wait for extant readers, or to require programmers to explicitly label the data being privatized.

As a proof of concept, we have implemented GMV on top of the TL2-like [6] "LLT" back end of the RSTM suite [19]. Experiments with microbenchmarks confirm that GMV eliminates starvation for long-running readers, yielding dramatically higher throughput than single-version systems for workloads that depend on such transactions.

We focus in this paper on the formal properties of GMV. We describe the algorithm, including its interface to the underlying STM system and its impact on privatization, in Section 2. In Section 3 we outline proofs of strict serializability, bounded space consumption, and wait-free readers. We also consider the impact of GMV on the liveness of writers. Section 4 summarizes the performance of our prototype implementation. We conclude in Section 5.

2 GMV Design

We refer to a transaction as a "reader" if it is known in advance to perform no updates to shared locations. Otherwise it is a "writer." On behalf of readers, and with limited cooperation from writers, GMV maintains four key data structures: a global timestamp variable, gt, that tracks the serialization order of writer transactions; an array ts of local timestamps, indexed by thread id; a history_table that holds values that have been overwritten by writers but may still be needed by active readers; and an array, hp, of "helping structures," also indexed by thread id. Variable gt can be shared with the underlying STM system (the *host*), if that system is timestamp-based. The history table, likewise, can be merged with the table of ownership records (Orecs) in the host, if it has such a table. Array hp is used to let the garbage collection process (invoked by writer threads) cooperate with reader transactions. Each reader records its history table inquiries in hp. If a writer needs to perform a potentially conflicting collection on a history list, it first completes the reader's request and stores the result in hp. GMV uses a type-preserving memory allocator for history nodes; this convention, together with the

monotonicity of timestamps, allows a reader to notice if its search has conflicted with a writer, and to retrieve the answer it was looking for from the helping array.

We characterize both GMV and the host as linearizable concurrent objects. The host provides methods for use by writers; GMV is oblivious to these. The host must also provide two methods to be called by GMV. GMV, for its part, exports four methods: two to be called by readers, the other two by the host. Readers make no direct calls to the host (Fig. 1). Our pseudocode assumes that memory is sequentially consistent, but it can easily be adapted to more relaxed machines.

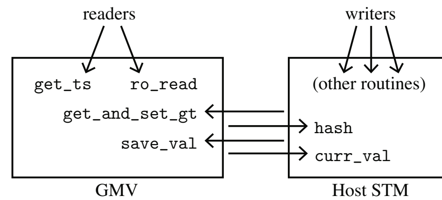

Fig. 1. GMV interface routines

GMV tracks overwritten values at word granularity, in a hash table keyed by memory address. Each bucket of the hash table is a dummy head node for a list of *history nodes* whose locations share a hash value. Each (real) node n in turn has three fields: a location loc, an old_value formerly contained in loc, and the global time (gt value) overwrite_time when this value was overwritten. A special-purpose, built-in garbage collector reclaims nodes that are no longer needed.

2.1 GMV–Host Interface

GMV provides two methods to be called by the host STM:

get_and_set_gt(): This method atomically increments gt and returns the new value. The host must guarantee that the serialization order of writers is consistent with the values returned. These values provide a well-defined meaning for "writer W serializes at time t," and "value v was written to location l at time t." Note that spurious calls to get_and_set_gt() are harmless: every committed writer must obtain a unique timestamp, but not every timestamp must correspond to a unique committed writer.

save_val(loc, old_value, overwrite_time): After calling get_and_set_gt(), and before allowing its thread to proceed with post-transaction execution, a writer must call this method for every location it has modified, passing the value returned by get_and_set_gt() as its overwrite_time. A call with a given location must not be made until all calls with a smaller overwrite_time and a location with the same hash value have already returned.

Code for these routines is trivial: get_and_set_gt performs a fetch_and_increment on gt and returns the result plus one; save_val writes its arguments into a newly allocated history node, which it then pushes, in the manner of a Treiber stack [24], onto the beginning of history list hash(l). We assume that the memory allocator employed by save_val tracks the total number of extant history nodes. Each writer checks this number at commit time. If it exceeds some predetermined threshold (we used $100K$ in our experiments), the writer invokes a garbage collection algorithm, described in Section 2.3.

GMV in turn requires two methods from the host:

hash(loc): Values returned by this function are used as indices into the history table. As noted above, the host must ensure that if two locations have the same hash value, calls to save_val will happen in timestamp order, even if they are made by different transactions.

curr_val(loc): GMV calls this method to obtain values not found in a history list. Its implementation must be wait-free. The host must guarantee that (1) if save_val(l, v, t) has been called (something that GMV of course can see), then the value v' returned by a subsequent call to curr_val(l) must have been written at some time $t' \geq t$, and (2) if curr_val(l) has returned v' and save_val(l, v'', t'') is subsequently called, then v' must have been written at some time $t' < t''$.

The implementation of curr_val depends on the nature of the host STM, but will often be straightforward. In a redo-log based STM, curr_val(l) can simply return the value at location l in main memory. In an undo-log based STM, it may need to access the log of some active writer W: it cannot require the reader to abort, nor can it wait for W to complete. It may also need to access the log of an active writer in a nonblocking STM [9, 12], where locations may be "stolen" without every having been written back to main memory. (The ordering requirements on calls to save_val are a bigger concern than save_val in nonblocking systems; we return to this subject in Section 3.3.)

2.2 Read-only Transactions

Aside from calls to curr_val, GMV handles reader transactions. At the beginning of reader R, executed by thread i, GMV stores the current global timestamp gt into local timestamp ts$[i]$. To read location l, R then calls ro_read(l) (Algorithm 1). When R commits, ts$[i]$ is set to infinity.

Algorithm 1. ro_read

```
Require: location l, thread id i
 1: h := history_table[hash(l)]
 2: v := reader_history_list_search(h, l, i)
 3: if v ≠ ⊥ then
 4:     return v
 5: c := curr_val(l)
 6: if h = history_table[hash(l)] then
 7:     return c
 8: v := reader_history_list_search(h, l, i)
 9: if v ≠ ⊥ then
10:     return v
11: else
12:     return c
```

At line 2 of ro_read, reader_history_list_search(h, l, i) looks for the last (oldest) node in history list h whose location field is l and whose overwrite time is greater (newer) than ts$[i]$. It returns $\perp$ if such a node does not exist. Code for this helper method appears in Algorithm 2. (The similar code in Algorithm 3 will be needed in Algorithm 4.) To enable helping by a garbage-collecting writer, ro_read maintains its current request—the location and time it's looking for—in hp$[i]$. During list traversal, if the reader sees a node with a larger than expected timestamp, it knows that a writer has interfered with its search, and that the answer it is looking for can be found in hp$[i]$.

Like other multiversion STM systems, GMV avoids reader transaction aborts by allowing them to "commit in the past." Where a writer transaction obtains its serialization time by calling get_and_set_gt when it is ready to commit, a reader obtains its serialization time by reading gt when it first begins execution. If reader R is long-running, it may

Algorithm 2. reader_history_list_search

Require: history list h, location l, thread id i

```
 1: v := ⊥;   n := h.next
 2: hp[i] := ⟨l, ts[i]⟩
 3: pt := ∞ {previous node timestamp}
 4: while n ≠ null do
 5:   if n→overwrite_time > pt then
 6:     {GC has interfered}
 7:     v := hp[i]
 8:     break
 9:   if n→overwrite_time ≤ ts[i] then
10:     {no further nodes will be useful}
11:     break
12:   if n→loc = l then
13:     v := n→old_value
14:   pt := n→overwrite_time
15:   n := n→next
16: hp[i] := ⊥
17: return v
```

Algorithm 3. GC_history_list_search

Require: hash value k, location l, time t

```
 1: while true do
 2:   v := ⊥;   n := history_table[k].next
 3:   pt := ∞ {previous node timestamp}
 4:   while n ≠ null do
 5:     nl := n→loc;   nv := n→old_value
 6:     nn := n→next
 7:     nt := n→overwrite_time
 8:        {read overwrite_time last}
 9:     if nt > pt then
10:       {another GC thread has interfered}
11:       continue while loop at line 1
12:     if nt ≤ t then
13:       {no further nodes will be useful}
14:       break
15:     if nl = l then
16:       v := nv
17:     pt := nt;   n := nn
18:   return v
```

serialize before a host of writer transactions whose implementations commit before it does. This "early serialization" resembles that of mainstream systems like TL2 [6], but multiversioning avoids the need to abort and restart read-only transactions that attempt to read a location that has changed since the transaction's start time. Early serialization stands in contrast to systems like RingSTM [23] and NOrec [5], which serialize readers at commit time, and to systems like TinySTM [22] and SwissTM [7], which dynamically update their "start time" in response to commits in other transactions, and may therefore serialize at some internal transactional read.

2.3 Garbage Collection

To avoid unbounded memory growth, history lists must periodically be pruned. If readers are never to abort, this pruning must identify and reclaim only those list nodes that will never again be needed. In GMV, a node may still be needed by reader i if it is the earliest node for its location that is later than $\mathsf{ts}[i]$. Nodes that do not satisfy this property for some thread i are reclaimed by the GC.

The core of the garbage collection algorithm appears in Algorithm 5. It is invoked from save_val, and can be executed concurrently by multiple writers. It has been designed to be lock free, and to preserve the wait freedom of readers. Writers synchronize with each other using a simplified version of the Harris [10] and Michael [14] lock-free list algorithm (simplified in the sense that insertions occur only at the head of the list). To support this algorithm, next pointers in history lists contain both a *count* and a *mark*.

The count, which is incremented whenever the pointer is modified, avoids the ABA problem. The mark indicates that a node is garbage and can be unlinked from the list; when set, it inhibits updating the pointer to link out the successor node.

As noted in Section 2.2, thread i begins a reader transaction by copying the global timestamp gt into $\mathsf{ts}[i]$. It ends by resetting $\mathsf{ts}[i]$ to infinity (maxint). To identify garbage nodes (Algorithm 6), we collect the entries in ts, sort them into descending order (with an end-of-list sentinel value), and then compare them to the timestamps of nodes in each history list via simultaneous traversal. The collect need not be atomic: nodes that transitioned from useful to garbage after the beginning of the scan may not necessarily be reclaimed, but the monotonicity of timestamps implies that anything that was garbage at the beginning of the scan is guaranteed to be recognized as such. If another writer finds that memory is getting low, it will call GC, discover nodes that can be freed, and keep the space bound by freeing them.

Algorithm 4. help_readers

Require: hash value k
1: **for** each thread id i **do**
2: $\quad x := \mathsf{hp}[i]$
3: $\quad$ **if** $x \neq \perp$ **then**
4: $\quad\quad \langle l, t\rangle := x$
5: $\quad\quad$ **if** $\mathsf{hash}(l) = k$ **then**
6: $\quad\quad\quad$ (void) CAS(&$\mathsf{hp}[i]$, x,
7: $\quad\quad\quad\quad$ GC_history_list_search(k, l, t))

To delete node n from a history list (having already read its predecessor's next pointer), we first mark n's next pointer. We then update the predecessor's next pointer to link n out of the list. We add n to a thread-local set of to-be-reclaimed nodes. Traversing the history list from head to tail, we effectively convert it to a tree. Any reader that is actively perusing the list will continue to see all useful successor nodes beyond ("above") it in the tree. Before we can actually reclaim the garbage nodes, however, we must ensure, via help_readers and GC_history_list_search (Algorithms 4 and 3) that no reader is still using them. We peruse the global helping array, hp. If we discover that reader R is searching for location l, and l hashes to the current history list, we complete R's search on its behalf, and attempt to CAS the result back into the helping array (in our pseudocode, this changes the type of $\mathsf{hp}[i]$, which is effectively a union). If the CAS fails, then either R has moved on or some other writer has already helped it. We can then safely reclaim our to-be-deleted nodes (moving them to a lock-free global free list), provided that we first update the timestamp in each so that a reader will recognize (line 5 of Algorithm 2) that it no longer belongs in the previous list.

2.4 Privatization Safety

It is generally recognized that any STM system for an unmanaged language must be *privatization safe* [13]. That is, if a transaction renders datum x accessible only to thread T, the STM system must ensure that (1) subsequent nontransactional writes of x by T cannot compromise the integrity of "doomed" transactions that may access x before aborting, and (2) delayed cleanup in logically committed transactions cannot compromise the integrity of nontransactional post-privatization reads of x by T.

We may safely assume that problem (2) is addressed by the host STM; the addition of GMV introduces no new complexity. Problem (1), however, is a challenge: if a privatizer writes to formerly shared data, and doesn't update the history table, an active

Algorithm 5. Garbage collection

```
1:  array st := sort(ts ∪ {−1}, descending)
2:  for k in hash range do
3:    node set G := find_garbage_nodes(k, st)
4:    node set U := ∅  {unlinked nodes}
5:    while true do
6:      p := &history_table[k]
7:      n := p→next
8:      pt := ∞ {previous node timestamp}
9:      while n ≠ null do
10:       nn := n→next
11:       nt := n→overwrite_time
12:       if nt > pt then
13:         {another GC thread has interfered}
14:         continue while loop at line 5
15:       if n ∈ G and ¬is_marked(nn) then
16:         if ¬CAS(&n→next, nn, mark(nn))
            then
17:           {another GC has interfered}
18:           continue while loop at line 5
19:       flag := false
20:       if is_marked(nn) then
21:         if CAS(&p→next, n, nn) then
22:           U += n
23:           flag := true
24:           if |U| ≥ U_MAX then
25:             help_readers(k)
26:             for n in U do
27:               n→overwrite_time := ∞
28:             reclaim all nodes in U
29:             U := ∅
30:         else
31:           {another GC has interfered}
32:           continue while loop at line 5
33:       if not flag then
34:         p := n;  pt := nt
35:       n := nn
36:     break
37:   help_readers(k)
38: for n in U do
39:   n→overwrite_time := ∞
40: reclaim all nodes in U
```

Algorithm 6. find_garbage_nodes

```
Require: hash val k, sorted time array st
1:  start_time := gt {global timestamp}
2:  while true do
3:    node set G := ∅ {garbage nodes}
4:    mapping[location⇒node] M := ∅
5:    i := 0;  n := history_table[k]
6:    pt := ∞ {prev. node timestamp}
7:    while n ≠ null and
      n→overwrite_time > start_time do
8:      {never reclaim nodes newer than
        start_time}
9:      n := n→next
10:   while st[i] ≠ −1 and n ≠ null do
11:     nl := n→loc;  nn := n→next

12:     nt := n→overwrite_time
13:         {read overwrite_time last}
14:     if nt > pt then
15:       {another GC has interfered}
16:       continue while loop at line 2
17:     if nt > st[i] then
18:       m := M[nl]
19:       if m ≠ null then
20:         G += m
21:       M[nl] := n;  n := nn
22:       pt := nt
23:     else
24:       i++;  M := ∅
25:   while n ≠ null do
26:     nn := n→next
27:     nt := n→overwrite_time
28:         {read overwrite_time last}
29:     if nt > pt then
30:       {another GC has interfered}
31:       continue while loop at line 2
32:     G += n;  n := nn
33:   break
34: return G
```

reader that needs to commit at some past time t may see the wrong value if calls curr_val. One possible solution is to require a privatizing writer to wait for all active readers to commit before it continues execution. This, of course, sacrifices nonblocking progress for writers (a subject to which we will return in Section 3.3). Even in a blocking system, it may induce an uncomfortably long wait. Alternatively, if the source program explicitly identifies the data being privatized, GMV could push the current values into the history table, where they would be seen by active readers. This option sacrifices the transparency of privatization. In a similar vein, if the compiler can identify data that *might* be sharable, it can instrument nontransactional writes to update the history list. This option compromises the performance benefit of privatization.

3 GMV Properties

In this section we sketch proofs of our claims of GMV safety, bounded space, and wait-free progress for read-only transactions ("readers"). We also consider the impact of GMV on the liveness of writers.

3.1 Safety

Theorem 1. *When GMV is correctly integrated into a strictly serializable host STM, the resulting STM remains strictly serializable.*

Proof. As described in Section 2.1, GMV requires the host STM, H, to ensure that (1) the serialization order of writer transactions is consistent with the values returned by get_and_set_gt, (2) a writer calls save_val(l, v, t) for every location it modifies, and (3) the calls for all locations with the same hash value occur in timestamp order. These rules ensure that history list nodes are ordered by timestamp, and that if $n_2 = \langle l, v_2, t_2 \rangle$ and $n_1 = \langle l, v_1, t_1 \rangle$ are consecutive nodes for location l ($t_2 > t_1$), then a reader transaction that sees v_2 at location l can correctly serialize at any time t such that $t_2 > t \geq t_1$.

Since nodes are removed from history lists only when there is no longer any reader transaction that can use them, the only remaining concern is for readers that call curr_val. In this case, as again described in Section 2.1, GMV requires H to ensure that any call to curr_val linearizes within H (1) after any method of H that calls save_val for the same location and a same or earlier timestamp, and (2) before any method of H that calls save_val for the same location and a later timestamp. These rules ensure that curr_val is called only when there is no appropriate history node, and that any writer that would cause curr_val to return a "too new" value calls save_val to create an appropriate history node first.

Taken together, the requirements on H ensure that a GMV reader sees exactly the same values it would have seen if executed as a writer in timestamp order within H. This in turn implies that the combined system remains strictly serializable. □

3.2 Space Consumption

Lemma 1. *In the wake of a call to Algorithm 5, started at time t, the total space consumed in history lists by nodes with timestamp less than t (denoted TS$_t$) is in $O(nm)$,*

where n is the total number of threads in the system, and m is the space consumed by a nontransactional, global lock-based program.

Proof. Algorithm 5 retains nodes that may be used by a concurrent reader. Therefore, for each location l, the GC retains a constant number of history list nodes for each currently active reader. We assume that the size of history_table (and hence of the extra head nodes) is bounded by $O(m)$. Since the total number of distinct locations is also in $O(m)$, and the total number of active readers is in $O(n)$, the total space for all nodes on all lists is clearly in $O(nm)$. □

Lemma 2. *Algorithm 5 is lock free*

Proof. We assume that the routines to allocate and reclaim list nodes are lock free. Given that history lists are noncircular, the traversal loops at Algorithm 3 line 4, Algorithm 5 line 9, and Algorithm 6 lines 7, 10, and 25 must all complete within a bounded number of steps. The remaining potential loops are the various **continue** statements: Algorithm 3 line 11; Algorithm 5 lines 14, 18, and 32; and Algorithm 6 lines 16 and 31. In most of these cases, execution of the **while true** loop continues when a GC thread encounters a node that has been reclaimed by some other thread (one whose timestamp appears larger than that of its predecessor); in these cases the system as a whole has made forward progress, and lock freedom is not endangered. The only tricky cases occur at Algorithm 5 lines 18 and 32, in the wake of a failed CAS. Here again the system as a whole has made progress: failure to mark or unlink a node indicates that some other thread has done so, and a marked node can be unlinked by any GC writer. □

Theorem 2. *The total space TS consumed by history lists is in $O(nm)$.*

Proof. Garbage collection will be started by any writer that discovers, at commit time, that the number of extant history nodes exceeds some predetermined threshold. Progress of the collection cannot be delayed or otherwise compromised by readers. Moreover any writer that attempts to commit before a GC pass has updated its statistics will also execute GC. By Lemma 2, so long as some thread continues to execute, some GC thread will make progress. By Lemma 1, a GC pass that starts at time t guarantees that TS_t is bounded by $O(nm)$. The only remaining question is then: what is the maximum value of $TS - TS_t$, the space that may be consumed, at the end of the GC pass, by history nodes that are unlinked but not reclaimed, or that have timestamp $\geq t$? This value is clearly the number of history nodes that may be generated by writers that are already in their commit protocol when the GC pass begins (TS_{added}), plus the number of nodes held by non-progressing GC threads (TS_{hold}, privatized at Algorithm 5 line 22). Since the number of writers is in $O(n)$, and the number of history nodes generated by any given writer is in $O(m)$, we know that TS_{added} is in $O(nm)$. For TS_{hold}, since each blocked GC may hold at most U_{MAX} nodes at a time (Algorithm 5 line 24), the total number of nodes held by non-progressing GC threads is in $O(n)$. It follows that $TS - TS_t$ is in $O(nm)$, and therefore so is TS. □

3.3 Liveness

Theorem 3. *GMV readers are wait free.*

Proof. Straightforward: by Theorem 2, the number of history nodes is bounded, and therefore so is the time spent traversing any given list in ro_read. We also require curr_val to be wait free. There are no other waits, loops, or aborts in the reader code. So all readers in GMV are wait free. □

By way of comparison, both SI-STM [21] and LSA [20] require readers to abort if the historical version they need has been reclaimed, so readers may in principle starve. JVSTM [3, 4] and SMV [17] never reclaim versions that may still be needed, but an active writer may create an unbounded number of history nodes for a reader to traverse. Systems that revert to inevitability for long-running readers are of course fundamentally blocking: a reader cannot start until active writers get out of the way.

Nonblocking writers. Ideally, we should like to be able to guarantee that if GMV were added to a lock-free (or obstruction-free) STM system, writers in the combined system would remain lock free (obstruction free). The GMV API functions are all lock free, which is certainly a good start: get_and_set_gt() is trivially lock free: its internal fetch_and_increment fails only if some other caller's succeeds. In a similar vein, calls to save_val() loop only when the Treiber-stack push fails because another thread's push succeeded. By Lemma 2, the garbage collection process called by save_val() is also lock free. Therefore save_val() is lock free.

Unfortunately, we must also consider the constraints we have placed on calls to these API functions. In particular, we have insisted that if $t_1 < t_2$ and hash(l_1) = hash(l_2), then any call of the form save_val(l_1, v_1, t_1) must occur before any call of the form save_val(l_2, v_2, t_2). This requirement is similar to asking transactions that modify locations with the same hash value to write their updates back to main memory in serialization order. It is not at all clear how a nonblocking system might do so. In particular, WSTM [8, 9] and MM-STM [12] (to our knowledge the only extant nonblocking word-based systems) both allow a transaction to "steal" an ownership record (Orec); values of locations that hash to that Orec may then be written back to memory out of order, up until the next time that the Orec is quiescent (if it ever is).

We believe we could obtain a (nonblocking) multiversion variant of WSTM or MM-STM by requiring the thread that steals an Orec to maintain the prefix of the history list corresponding to that Orec's locations. Method ro_read would begin by consulting the Orec: if quiescent, it would consult the usual history list; otherwise, it would first consult the stealer's list prefix. This solution would require that GMV be integrated into the underlying system in a way that no longer merits the term "generic." We leave the details to future work.

4 Performance of a Proof-of-Concept Implementation

We implemented a proof of concept system, GMV+, for GMV. This implementation is based on the LLT back end, a TL2 [6]-like STM, in the RSTM suite [19].

GMV+ differs from GMV only in the addition of a "fast path" for garbage collection. This path reclaims only the tails of history lists, in a region known to be ignored by all still-running readers, thereby eliminating the need for helping. If memory consumption is still beyond the preset threshold after execution of the fast path, GMV+ returns to

execute the normal "slow path" GC algorithm, with helping. In our experiments, the slow path was very rarely needed.

We tested GMV+ on a two-processor Intel Xeon E5649 machine. Each processor has 6 cores running at 2.53 GHz, and 2 hardware threads per core. Each core has 32 KB of L1 D-cache and 256 KB of L2 cache; the cores of a given processor share 12 MB of on-chip L3 cache. Microbenchmark results indicate that the maintenance of history lists increases the overhead of writers by approximately 50%. In return, multiversioning reaps significant benefits when the workload has long-running readers. We modified a hash table microbenchmark that performs lookup, insert, and remove operations, to also 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.

Figure 2 (top) present results for a read-heavy test with sum, lookup, and update (insert and delete) operations in a ratio of 1:79:20. We compare the throughput (transactions/second) of LLT, GMV+, and two variants of the simpler NOrec algorithm [5]. 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. Other threads read this lock; if it is held they abort, and wait to retry.

When running our microbenchmark, GMV+ outperforms the other tested algorithms, with speedup out to the full count of hardware threads. While inevitability avoids starvation of readers, it also limits scalability: neither algorithm with inevitability speeds up with additional threads.

We also evaluate GMV+'s performance on a modified version of the "Vacation" benchmark from the STAMP suite [15]. Vacation simulates a concurrent travel inquiry / reservation system. Most threads, as in the original version, repeatedly perform read / write / update operations on price tables (for cars, flights and rooms), and read / write operations on the reservation table. At the same time, we add a dedicated "checker" thread that periodically runs a transaction to checksum the reservation table. Note that in contrast to the hash table microbenchmark, here long-running read-only transactions are confined to a single thread. Overall system throughput is displayed in the bottom half of Figure 2.

We run this benchmark with 4 queries per normal transaction and 65536 initial relations in each price table. 98% of normal transactions are for reservations; the other 2% update price tables. The benchmark's "query range" parameter is set to 60% for normal transactions, which the application's authors consider "high contention." We run the checker every 100 ms in this test. Without inevitability, checker transactions routinely starve in both LLT and NOrec. We omitted results for these configurations in the figure. With inevitability, the checker can almost always complete within 100 ms. It usually completes within this interval for GMV+ as well, at least at low thread counts.

Overall transaction throughput for GMV+ is roughly 20% higher than for LLT with inevitability, presumably because the checker thread, when running, does not exclude concurrent writers. Throughput peaks at 12 threads (the number of cores) on the

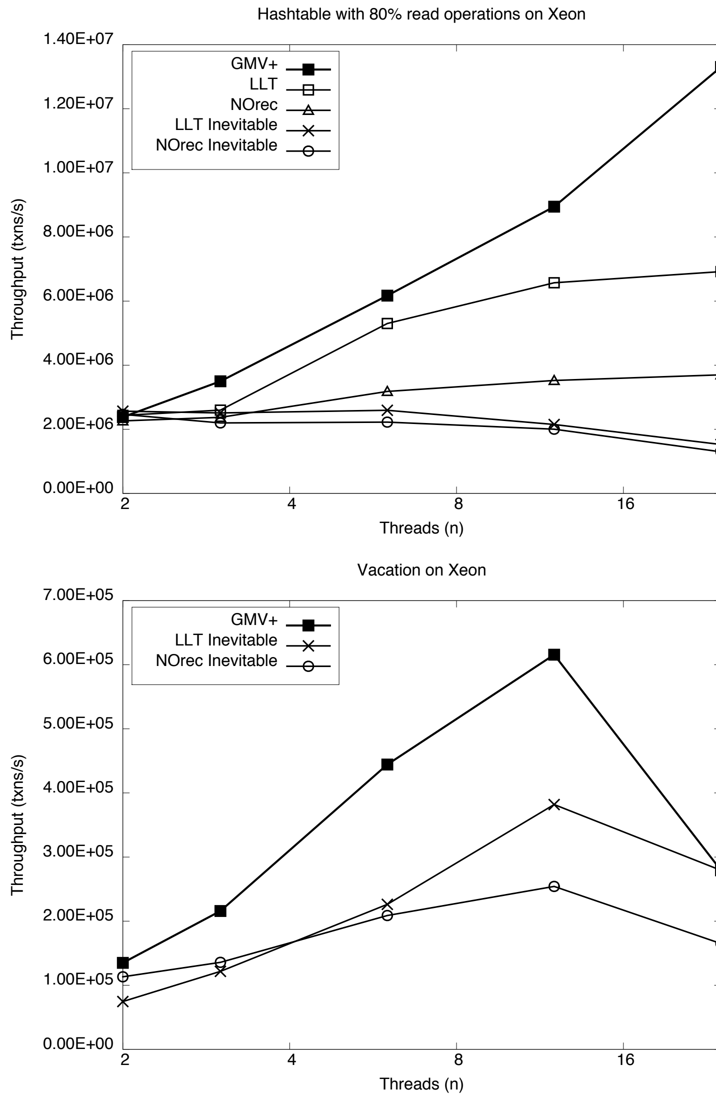

Fig. 2. Throughput of GMV+ for hash table (top) and augmented Vacation (bottom)

testing machine. The default scheduling discipline places successive threads on alternating processors, so inter-chip communication is occurring even at low thread counts. In this situation NOrec's scalability is limited by contention on the lock that serializes writer commits.

Performance results for GMV+ confirm that multiversioning is an attractive alternative to inevitability for applications with long-running read-only transactions. Multiversioning allows long-running readers 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.

5 Conclusions

We have proposed a generic multiversion STM system, GMV. Unlike previous integrated systems, it can be layered on top of most existing word-based STM. To the best of our knowledge, GMV is the first STM system to combine bounded space consumption with guaranteed wait freedom for read-only transactions. It neither requires nor interferes with automatic garbage collection. Privatization can be ensured—without compromising wait freedom for readers—either by blocking writers or by requiring that programmers explicitly identify the data being privatized.

We also described a proof-of-concept implementation of GMV. With roughly 50% overhead to maintain history lists, our implementation eliminates reader starvation, and generates up to 2× speedup on workloads with long-running readers. With further implementation effort, the instrumentation overhead could probably be reduced, but for small transactions it will always be higher than the baseline. Topics for future work include (1) integration with nonblocking word-based STM; (2) automatic mechanisms to choose when a read-only transaction should use the history lists (as opposed to acting as a writer); and (3) a mechanism to choose (on a global basis), when writers should maintain the history lists.

Acknowledgment. We are grateful to the anonymous referees for identifying several bugs in the pseudocode, and for prodding us to clarify our thinking on the issue of nonblocking writers.

References

1. Aydonat, U., Abdelrahman, T.: Serializability of Transactions in Software Transactional Memory. In: 3rd ACM SIGPLAN Wkshp. on Transactional Computing, Salt Lake City, UT (February 2008)
2. Bieniusa, A., Fuhrmann, T.: Consistency in Hindsight: A Fully Decentralized STM Algorithm. In: Proc. of the 24th Intl. Parallel and Distributed Processing Symp., Atlanta, GA (April 2010)
3. Cachopo, J., Rito-Silva, A.: Versioned Boxes as the Basis for Memory Transactions. Science of Computer Programming 63(2), 172–185 (2006)
4. Cachopo, J., Rito-Silva, A.: Versioned Boxes as the Basis for Memory Transactions. In: Proc., Wkshp. on Synchronization and Concurrency in Object-Oriented Languages, in conjunction with OOPSLA 2005, San Diego, CA (October 2005)
5. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: Streamlining STM by Abolishing Ownership Records. In: Proc. of the 15th ACM Symp. on Principles and Practice of Parallel Programming, Bangalore, India (January 2010)
6. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: Proc. of the 20th Intl. Symp. on Distributed Computing, Stockholm, Sweden (September 2006)
7. Dragojević, A., Guerraoui, R., Kapałka, M.: Stretching Transactional Memory. In: Proc. of the SIGPLAN 2009 Conf. on Programming Language Design and Implementation, Dublin, Ireland (June 2009)

8. Fraser, K., Harris, T.: Concurrent Programming Without Locks. ACM Trans. on Computer Systems 25(2), article 5 (May 2007)
9. Harris, T., Fraser, K.: Language Support for Lightweight Transactions. In: OOPSLA 2003 Conf. Proc., Anaheim, CA (October 2003)
10. Harris, T.L.: A Pragmatic Implementation of Non-Blocking Linked-Lists. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, p. 300. Springer, Heidelberg (2001)
11. Keidar, I., Perelman, D.: On Avoiding Spare Aborts in Transactional Memory. In: Proc. of the 21st ACM Symp. on Parallelism in Algorithms and Architectures, Calgary, AB, Canada (August 2009)
12. Marathe, V.J., Moir, M.: Toward High Performance Nonblocking Software Transactional Memory. In: Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming, Salt Lake City, UT (February 2008) Expanded version available as TR 932 Dept. of Computer Science, Univ. of Rochester (March 2008)
13. Marathe, V.J., Spear, M.F., Scott, M.L.: Scalable Techniques for Transparent Privatization in Software Transactional Memory. In: Proc. of the 2008 Intl. Conf. on Parallel Processing, Portland, OR (September 2008)
14. Michael, M.M.: High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In: Proc. of the 14th ACM Symp. on Parallel Algorithms and Architectures, Winnipeg, MB, Canada (August 2002)
15. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: Proc. of the 2008 IEEE Intl. Symp. on Workload Characterization, Seattle, WA (September 2008)
16. Napper, J., Alvisi, L.: Lock-Free Serializable Transactions. Technical report TR-05-04, Dept. of Computer Sciences, Univ. of Texas at Austin (February 2005)
17. Perelman, D., Byshevsky, A., Litmanovich, O., Keidar, I.: SMV: Selective Multi-Versioning STM. In: Proc. of the 25th Intl. Symp. on Distributed Computing, Rome, Italy (September 2011)
18. Perelman, D., Fan, R., Keidar, I.: On Maintaining Multiple Versions in STM. In: Proc. of the 29th ACM Symp. on Principles of Distributed Computing, Zurich, Switzerland (July 2010)
19. Reconfigurable Software Transactional Memory Runtime. Project web site, `code.google.com/p/rstm/`
20. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 284–298. Springer, Heidelberg (2006)
21. Riegel, T., Fetzer, C., Felber, P.: Snapshot Isolation for Software Transactional Memory. In: 1st ACM SIGPLAN Wkshp. on Transactional Computing, Ottawa, ON, Canada (June 2006)
22. Riegel, T., Fetzer, C., Felber, P.: Time-based Transactional Memory with Scalable Time Bases. In: Proc. of the 19th ACM Symp. on Parallelism in Algorithms and Architectures, San Diego, CA (June 2007)
23. Spear, M.F., Michael, M.M., von Praun, C.: RingSTM: Scalable Transactions with a Single Atomic Instruction. In: Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures, Munich, Germany (June 2008)
24. Treiber, R.K.: Systems Programming: Coping with Parallelism. RJ 5118, IBM Almaden Research Center (April 1986)