Versioned Programming: A Simple Technique for Implementing Efficient, Lock-Free, and Composable Data Structures

Yang Zhan Donald E. Porter
Stony Brook University
{yazhan, porter}@cs.stonybrook.edu

Abstract

This paper introduces versioned programming, a technique that can be used to convert pointer-based data structures into efficient, lock-free implementations. Versioned programming allows arbitrary composition of pointer modifications. Taking linked-lists as an example, VLISTs, or versioned lists, support features missing in other lock-free implementations, such as double linking and atomic moves among lists.

The main idea of versioning is to allow different versions of a nodes exist at the same time such that each thread can pick the appropriate version and has a consistent view of the whole data structure. We present a detailed example of VLISTs, simple enough to include all code inline. The paper also evaluates versioned tree implementations.

We evaluate versioned programming against several concurrency techniques. With a modest number of writers, versioned programming outperforms read-log-update, which locks nodes. VLIST out-perform lists with SwissTM, a high-quality STM, showing the value of trading some programmer-transparency for performance. Composability hurts performance compared to a non-composable, hand-written lock-free algorithm. Using the technique described in this paper, application developers can have both the performance scalability of sophisticated synchronization techniques with functionality and simplicity comparable to coarse locks.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords Concurrent Programming, Lock-Free Data Structures, Versioned Programming

SYSTOR '16, June 6–8, 2016, Haifa, Isreal.

http://dx.doi.org/10.1145/2928275.2928285

1. Introduction

Software developers need better tools to write concurrent programs for new generations of multi-core hardware. Many programs adopt locking to share data structures safely, at a cost to performance scalability. Lock-free data structures can permit more concurrency and have strong progress guarantees under write contention, but with restricted functionality. Not all data structures have lock-free variants.

Existing lock-free data structures, including the state-of-the-art Harris-Michael list [15, 26], can safely add or remove individual items from a list, but cannot compose multiple operations safely. For instance, if an application moves an item from one list to another, the list implementation can atomically remove the item from the first list and atomically add the item to the second list, but there will be a period during which the item is either on both lists, or on neither list. If the application has an invariant that requires the item to be on exactly one list, the only safe option is to use a general-purpose synchronization primitives, e.g., locks or transactional memory. The inability to safely compose complex list operations prevents some applications from enjoying the performance benefits of current lock-free lists.

Our insight into the problem is that, to over-simplify slightly, the art of designing a lock-free data structure is the art of mapping pointer manipulations onto a single, atomic instruction. Composing arbitrary data structure mutations requires mapping an arbitrary number of pointer manipulations onto a single atomic instruction, which is difficult to do efficiently. Best-effort hardware transactional memory (HTM) [1, 32] can compose multiple pointer manipulations into one atomic operation, but current hardware transactions can fail for a wide array of architectural and microarchitectural reasons, leaving programmers with no guarantees that a given transaction will commit, even in the absence of contention. Thus, any practical lock-free data structure cannot use HTM by rote, but must be able to fall back to software.

This paper introduces versioned programming, a technique that addresses the dilemma between functionality and scalability. Versioned programming encapsulates the complexity of manipulating an arbitrary number of pointers while maintaining lock-free progress guarantees and good

performance scalability. Versioned programming can be roughly thought of as an adaptation of multi-version concurrency control (MVCC) [3, 31] or optimistic concurrency control (OCC) [21] for use in pointer-based data structures. The technique is simple enough to present and explain all non-boilerplate code for a both the data structure library and its application to a doubly-linked list implementation, which primarily involves wrapping pointer manipulation and adding a retry loop for writers. For generality, we also use this technique to implement a lock-free, balanced tree.

Finally, we note that the difficulty of having both scalability and ease-of-use in all cases has led to a number of programming models that make explicit trade-offs. Read-copy update (RCU) [25] and read-log update (RLU) [24], make data structures scalable in special-cases, such as read-mostly access patterns, but offer weaker progress guarantees in the general case, such as with a non-trivial number of writers. Software transactional memory (STM) implementations sacrifice performance for generality. Linked lists are a long-known pathological case for TM, inducing false conflicts on pointers that all threads have to traverse [29]. Thus, high-performance TM implementations often suspend the transaction for list traversals with open nesting [28, 29] or transactional boosting [19]. Section 4 also outlines how the versioned programming library could use HTM internally to improve its performance. Finally, we note that the success of the Java Concurrency package indicates that applications can benefit from concurrent data structures without transforming the entire application to use transactions.

We evaluate the versioned programming for a linked list and balanced tree on a 48-core machine; compare to RCU, RLU, and SwissTM; and draw several insights about the performance scalability of both versioning and alternative approaches to constructing concurrent data structures.

- With heavy writes and modest thread counts, our versioned list, called a VLIST, is second only to a Harris-Michael list.
- Composability has a cost for simple operations, and a non-composable Harris-Michael list outperforms a VLIST.
- On a list move benchmark, which exercises composability, VLISTs outperform RLU and SwissTM at 16 or more threads.

Thus, versioned programming strikes a good balance between functionality and performance scalability.

2. Background on lock-free lists

Because this paper uses linked-lists as a running example, this section first reviews the Harris-Michael lock-free list [15, 26]—the *de facto* lock-free list algorithm. The Harris-Michael lock-free list is used in the Java Concurrency Package's `ConcurrentSkipListSet` and `ConcurrentSkipListMap` classes.

The Harris-Michael algorithm is restricted to singly-linked lists, and in these examples, we assume the list is sorted by key. We assume the interface for all list implementations includes three functions: search, insert, and delete.

The `search` function is fairly straightforward: reader threads simply follow a series of `next` pointers until they arrive at the desired list node. The insert and delete functions, therefore, must modify the list such that searches never dereference a bad pointer value.

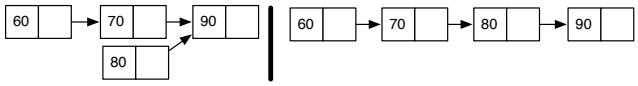

Figure 1. Key steps to insert a node in a Harris-Michael list.

The implementation of `insert` is illustrated in Figure 1. The list is first traversed to identify where to insert the node. The new list node, for key 80 in the Figure, is allocated and populated with an appropriate `next` pointer. Finally, the `next` pointer of the previous node (70) is set with an atomic compare-and-swap (CAS) instruction. If the CAS fails, the insert must be retried. This CAS ensures that no other node was inserted after node 70 in the list, but there is another race condition that requires additional effort to prevent. As node 80 is being inserted into the list, another thread could delete node 70 from the list (Figure 2). For this reason, a number of lock-free linked list algorithms require a double-compare-and-swap (DCAS) instruction, which has not been implemented on a processor since the Motorola 68k for performance reasons [14, 23].

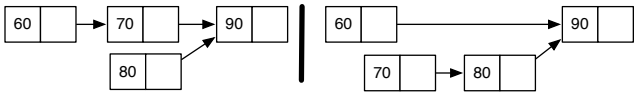

Figure 2. Insertion race condition with a single CAS.

Therefore, the challenging part of this and other lock-free list algorithms is implementing `delete` such that concurrent operations work correctly. The key insight of Harris's original algorithm is separating *logical* deletion from *physical* deletion, as illustrated in Figure 3. Logical deletion is performed by setting the next pointer of a deleted node (70) to a numerically distinct value that still allows traversal. For example, if list nodes are word-aligned, one could set the low bit of the pointer to indicate logical deletion. This allows a reader or inserter to continue traversing the list, as well as reliably test for logical deletion.

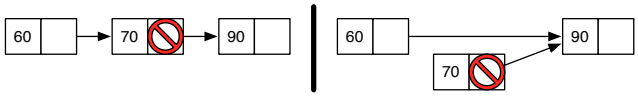

Figure 3. Key steps to delete a node from an HM list.

After a node is logically deleted, it is physically removed from the list by changing the `next` pointer of the previous

node with a CAS operation. Again, whether a reader is traversing the list before or after the CAS, it can safely traverse the list and filter out any logically deleted nodes.

The final issue is determining when the memory for a deleted list node can be reclaimed safely. Note that even after physical deletion, a thread may still refer to this node for some time. Harris's original algorithm proposed reference counting garbage collection; Michael refined the algorithm to incorporate other strategies including hazard pointers [27], which explicitly track active object references by readers. In either approach, a deleted list node must persist until there is no potential thread reading the node.

Limitations. The Harris-Michael algorithm is elegant in its simplicity, but functionally limited. Concurrent insertions and removals are handled safely, but programmers cannot compose larger operations, such as moving an item from one list to another. Another issue is that Harris-Michael lists are singly-linked; doubly-linked lists require a general-purpose synchronization mechanism.

RCU and RLU lists. Read-copy update (RCU) linked lists [25] allow limited composition with a Harris-Michael-like design. RCU is designed to improve the scalability of read-mostly data structures. RCU eliminates read locking, but serializes writers with locks. RCU linked lists inherit many of the limitations discussed above: no support for doubly-linked lists, and the inability to move an embedded list node to another list until there are no potential readers.

Read-Log-Update (RLU) [24] is a extension of RCU that simplifies the effort to use RCU, allows multiple writers and composition of operations. We note that there are some structural similarities between RLU and versioned programming, which were developed independently and concurrently. One major difference is that RLU uses locks, which lead to weaker progress guarantees and lower performance under contention. Any lock-based approach, including RLU, will inherit an unsavory trade-off between complexity and performance in how fine or coarse to make the locks. A key contribution of versioned programming is making such a design lock-free and improving performance with larger thread counts and write contention.

3. Overview

This section presents the intuition of versioned programming in a lock-free linked list, called a VLIST, or versioned list. The versioning design applies multi-version concurrency control [3], common in databases and even a few STMs [4], to the pointers in a data structure. Making this versioning technique suitable for use in a stand-alone data structure is a key contribution of this paper.

In a nutshell, VLISTs work as follows. Each node in a VLIST contains multiple versions of its contents, called *slots*, organized into a simple list sorted from most recent to oldest. Slots are versioned according to a monotonically

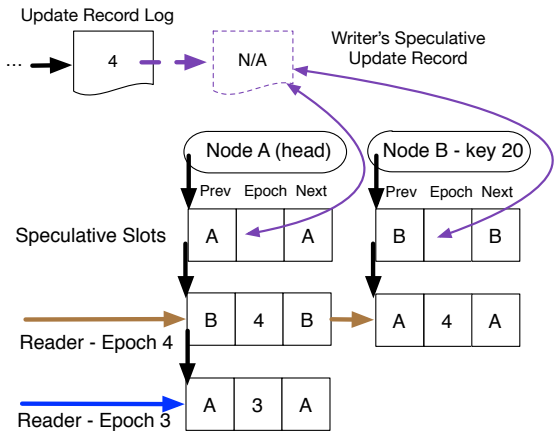

Figure 4. Two threads with different epoch reading a list, and one speculatively deleting node B. Each thread sees a view of the list appropriate to the point in logical time in which it is serialized. The reader with epoch 4 sees Node B with key 20, and the reader with epoch 3 does not.

increasing *epoch*, or logical timestamp. Each write critical section increases the epoch. Unrelated data structures may have different epoch counters; an epoch counter must be shared among data structures that require safe composition, such as atomically moving an item from one list to another. In a doubly-linked list, each slot includes the previous and next pointers, as well as the epoch.

Threads pick an epoch to use while traversing the list—generally the most recent epoch at the start of a critical region—and see a consistent view of the list's state at that point. As a reader traverses each list node, it walks each node's internal list of slots, and selects the most recent slot that is not newer than the thread's epoch.

Figure 4 illustrates a VLIST. One reader attempts to read the list at epoch 3. The reader starts at the list head, node A, and gets the most recent slot as of epoch 3. The slot indicates only node A is present in the list at epoch 3. Another reader tries to read the list at epoch 4 and selects the slot with epoch 4. The slot shows that node B is added to the list at epoch 4. Therefore, the reader is able to see node B.

Writers insert *speculative* slots into list nodes, which are ignored by readers until the writer adds an *update record* to a shared log of updates. Speculative slots allow writers to detect unsafe, concurrent writes from other threads. The update record log creates a total order for all list modifications, and also linearizes (or commits) an arbitrarily large set of consistent updates with a single atomic compare-and-swap (CAS) instruction. Disjoint nodes can be modified concurrently, and can be serially added to the log in quick succession.

In Figure 4, the top slot of each node is a speculative deletion of node B from the list. This update record can be applied with a single CAS to the end of the update log.

The VLIST implementation efficiently garbage collects deleted nodes, old slots and out-of-date records using a

quiescence-based reclamation scheme [25], which ensures that no internal data structures are reclaimed concurrently with access by a thread. As with any lock-free, dynamic data structure, versioned data structures require a lock-free memory allocator for end-to-end lock-freedom. Petrank et al. [30] prove that the composition of an otherwise lock-free data structure with a lock-free allocator will be lock-free, and thus these issues can be treated as orthogonal.

4. Versioned list algorithm

This section details the VLIST design. We begin this section with an intuitive overview of the algorithm, followed by a detailed description of the reader and writer code in lists sorted by keys, and conclude the section with a discussion of certain design issues. Because the algorithm is so simple, we include all non-boilerplate code in the text as figures, using the C syntax of our prototype implementation. We do omit the implementation of `move`, which is pattern-matched from `insert` and `remove`, and available in the released code at `https://github.com/oscarlab/versioning`.

Versioned list nodes. As illustrated in Figure 4, each node in a VLIST list stores multiple versions of the data structure's pointers. Each VLIST node includes an internal list of *slots*, which include different addresses the next and previous pointers have pointed to over the life of the node. Slots are sorted in reverse chronological order, and valid slots are assigned a logical timestamp, or *epoch*. Invalid slots, which represent a failed speculation, are ignored, as are uncommitted speculative slots.

When a thread begins traversal of a VLIST, the thread reads the most recently committed epoch value and stores this as the *thread epoch*. Intuitively, a forward traversal of a VLIST would start at the head node, read the most recent slot from the head node with an epoch equal to or older than the thread epoch, dereference the next node pointer, and repeat this process until the thread arrived back at the head node.

Optimistic writers. The VLIST algorithm is optimistic, in that it allows concurrent writers to speculatively update nodes in the list. Writers allocate slots to nodes for speculative modifications, which are ignored by readers until the entire operation is validated for safety, and then linearized. Pending modifications are logged in an *update record*, and update records are applied to an ordered log with a single, atomic compare-and-swap. These update records primarily serve the purpose of consolidating updates across a range of nodes to a single linearization point; these update records are eventually garbage collected and replaced with epoch values in the slots (described further in §4.2). Deleted nodes, obsolete slots, and update records are garbage collected using a quiescence-based reclamation scheme [25].

When a writer is ready to commit its modifications, the writer compares its update record to all update records added after the writer began its critical region. The writer compares

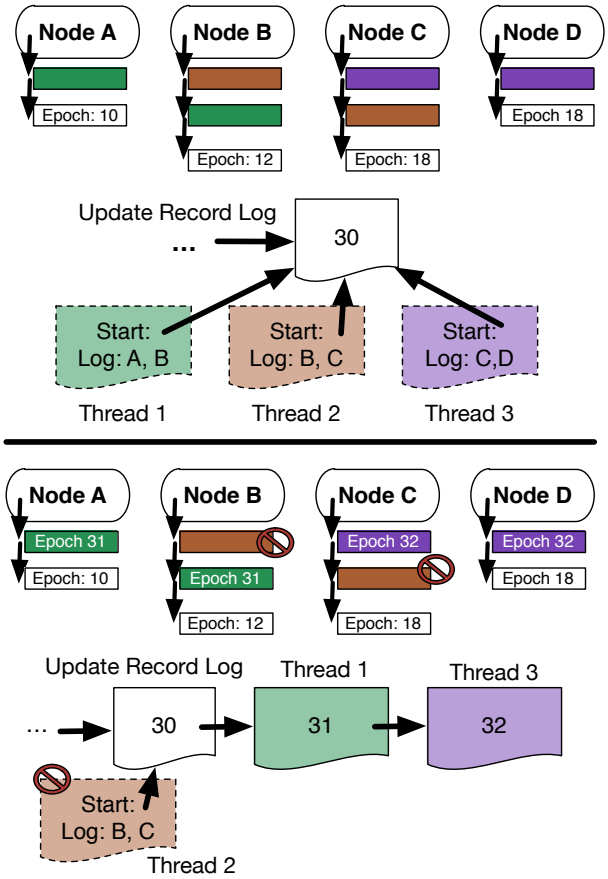

Figure 5. Three concurrent, speculative writers in a VLIST. Thread 1 is speculatively adding slots to nodes A and B, Thread 2 is adding slots to B and C, and Thread 3 is adding a slots to C and D. All slot addition proceeds in parallel, but updates are linearized by a CAS on the tail of the update record log. Before adding a record, intervening records must be checked: Thread 2 will observe a conflict with Thread 1 and retry, whereas Threads 1 and 3 will commit. Readers ignore speculative or discarded entries until they are committed and assigned an epoch value.

the list of nodes has modified to the list in each intervening record in order to detect any conflicting updates to the same node. If the writer detects conflicts, it marks all speculative slots as invalid (reserved epoch value 1), and retries.

If there are no conflicting updates between reading the list and applying updates, the writer adds its update record to the tail of the update log with an atomic compare-and-swap (CAS) of the last entry's next pointer. This is the *linearization point* of the VLIST algorithm. If the CAS fails, the writer continues walking the update record list until it detects a conflict, or the CAS succeeds.

If the list nodes are uncontended, multiple write critical sections can execute in parallel and then commit in quick succession, described in detail in Section 4.2. If two writers conflict, one has to retry its modifications.

```
1  struct vlist_slot {
2    uint64_t epoch;
3    struct vlist_node *prev, *next;
4    struct vlist_slot *slot_next;
5    struct update_record *pt;
6  };
7  struct vlist_node {
8    int value;
9    struct vlist_slot *slots;
10 };
11 struct update_record {
12   uint64_t epoch;
13   int count;
14   struct vlist_head *heads[VLIST_ENTRIES_PER_TASK];
15   struct slot_t *slots[VLIST_ENTRIES_PER_TASK];
16   struct update_record *rec_next;
17 };
18 struct vlist_info {
19   uint64_t epoch;
20   struct update_record *rec, *new_rec;
21 };
```

Figure 6. VLIST data types. The `vlist_node` represents a node in the list, and includes a list of `vlist_slot` structures. An `update_record` stores modifications, and is added to the log using an atomic CAS. A `vlist_info` structure stores per-thread bookkeeping for the algorithm.

```
1  int search(struct vlist_node *head, int key) {
2    read_cs_enter();
3    for (curr = read_slot(head)->next;
4         curr != head; curr = read_slot(curr)->next)
5      if (curr->key >= key)
6        break
7    ret = (cur->key == key)
8    read_cs_exit();
9    return ret;
10 }
11 struct slot_t *read_slot(struct vlist_node *node) {
12   for (it_slot = node->slots; it_slot != NULL;
13        it_slot = it_slot->slot_next) {
14     if (it_slot->epoch == INDIRECT_EPOCH)
15       slot_epoch = it_slot->pt->epoch;
16     else
17       slot_epoch = it_slot->epoch;
18     if (slot_epoch != INACTIVE_EPOCH &&
19         slot_epoch <= thread->epoch)
20       break;
21   }
```

Figure 7. A VLIST search implementation.

Figure 5 illustrates three concurrent writer threads. Each thread adds speculative slots to different nodes in parallel. Once the thread is ready to commit, it checks for intervening update records in the log and checks for an intersection with the set of written nodes. In this example, suppose Thread 1 adds its record first. Thread 3 has no conflict and can commit; Thread 2 conflicts with Thread 1 on Node B and must abandon its modifications and retry.

VLISTS **are lock-free.** Readers can always make progress, and at least one writer can always make progress. A failed or delayed writer cannot obstruct other writers, because all speculative modifications are ignored.

Explicit critical sections. In order to operate on a VLIST, a thread must manipulate the list in an explicit read or write critical region. We primarily envision use of VLISTs encapsulated within a library, similar to the Java concurrent list libraries, but extended with `atomic_move` and other functions that require composition. Most of this section describes the algorithm using library-internal, low-level operations, which could also be exposed to a sophisticated application developer who understands how to compensate for side effects outside of the list on a retry.

4.1 List Readers

The types used in all of the VLIST examples are listed in Figure 6. The `vlist_info` structure stores per-thread bookkeeping, a `vlist_node` is a list node, which contains an internal list of `vlist_slot` structures. An `update_record` stores modifications by a writer, which are committed with a CAS to the tail of the `update_record` list. VLISTs reserve two epochs. `INACTIVE_EPOCH` indicates pending or failed updates that are ignored by readers, and `INDIRECT_EPOCH` requires readers to read the epoch from `update_record`. The value `VLIST_ENTRIES_PER_TASK` sets a configurable upper bound on the number of modifications that can be composed in one update record.

Figure 7 shows a VLIST search implementation. Compared to another linked list, the implementation is very straightforward, except that the critical section is demarcated with `read_cs_enter()` and `read_cs_end()`, and that dereferences of the `next` pointer are first wrapped with the `read_slot()` helper function.

The `read_slot` function walks the internal list of slots associated with each list node. These slots are sorted from most recent to oldest. Line 18 skips speculative entries in the list. If `read_slot` encounters a slot with an indirect epoch (the epoch is not known until the update record is committed), the epoch is read from the update record (Line 15). The key invariant in this search is that the reader needs to see the most-recently created slot with an epoch less than or equal to the thread's epoch (Line 19)—ensuring a consistent traversal of the list.

INVARIANT 4.1. *The list of slots in a node are sorted in reverse epoch order (newest, or highest values, first), excluding failed speculative writes, which are ignored.*

Starting and Ending Read Critical Sections. The primary task to start a read critical section (`read_cs_enter()`, in Figure 8), is to walk the list of `update_records` starting from the last observed record, until the tail is reached. Thus, a reader traverses the structure using the latest epoch value.

Ending a critical section simply involves calling the quiescence library, which we adopt from Hart et al. [16]. Each thread announce a quiescent state after certain number of operations, which helps other threads decide whether they can safely free objects in free lists.

```
1  static inline void set_thread_epoch(void) {
2    next = thread->rec;
3    epoch = next->epoch;
4    while (next = next->rec_next) {
5      epoch++;
6      if (next->epoch == INACTIVE_EPOCH)
7        next->epoch = epoch;
8      thread->rec = next;
9    }
10   thread->epoch = epoch;
11 }
12 static inline void read_cs_enter(void) {
13   set_thread_epoch();
14 }
15 static inline void read_cs_exit(void) {
16   /* Quiesce */
17 }
18 static inline void write_cs_enter(void) {
19   set_thread_epoch();
20   thread->new_rec = new_record();
21   thread->new_rec->epoch = INACTIVE_EPOCH;
22 }
23 static int write_cs_exit(void) {
24   ret = 0; new_rec = thread->new_rec;
25   epoch = thread->rec + 1;
26   if (new_rec->count > 0) {
27     while (true) {
28       for (cursor = thread->rec; cursor;
29            cursor = cursor->rec_next) {
30         for (i = 0; i < cursor->count; i++)
31           for (j = 0; j < new_rec->count; j++)
32             if (new_rec->heads[i] == cursor->heads[j]) {
33               ret = 1; goto out;
34             }
35         epoch += 1; thread->rec = cursor;
36       }
37       if (CAS(&thread->rec->rec_next, NULL, new_rec)) {
38         new_rec->epoch = epoch;
39         for (i = 0; i < new_rec->count; i++)
40           new_rec->slots[i]->epoch = epoch;
41         break;
42       }
43     }
44   } else free_record(new_rec);
45 out:
46   if (ret == 1) {
47     for (i = 0; i < new_rec->count; i++)
48       new_rec->slots[i]->epoch = INACTIVE_EPOCH;
49     free_record_later(new_rec);
50     write_cs_enter();
51   } else
52     /* Quiesce */
53   return ret;
54 }
```

Figure 8. VLIST reader/writer helper functions.

```
1  int insert(struct vlist_node *head, long key) {
2    newnode = new_node(key);
3    write_cs_enter();
4    do {
5      for (prev = head, prev_slot = read_slot(head);
6           (curr = prev_slot->next) != head;
7           prev = curr, prev_slot = read_slot(prev))
8        if (curr->key >= key) break;
9      if (curr != head && curr->key != key) {
10       ret = 1;
11       curr_slot = read_slot(curr);
12       add_slot(prev, prev_slot->prev, newnode);
13       add_slot(newnode, prev, curr);
14       add_slot(curr, newnode, curr_slot->next);
15     } else ret = 0;
16   } while (write_cs_exit());
17   if (!ret) free_node_later(newnode);
18   return ret;
19 }
20 int delete(struct vlist_node *head, long key) {
21   write_cs_enter();
22   do {
23     for (prev = head, prev_slot = read_slot(head);
24          (curr = prev_slot->next) != head;
25          prev = curr, prev_slot = read_slot(prev))
26       if (curr->key >= key) break;
27     if (curr != head && curr->key == key) {
28       ret = 1;
29       curr_slot = read_slot(curr); next = curr_slot->next;
30       next_slot = read_slot(next);
31       add_slot(prev, prev_slot->prev, next);
32       add_slot(curr, curr, curr);
33       add_slot(next, prev, next_slot->next);
34     } else ret = 0;
35   } while (write_cs_exit());
36   if (ret) free_node_later(curr);
37   return ret;
38 }
39 void add_slot(struct vlist_node *head,
40               struct vlist_node *prev,
41               struct vlist_node *next) {
42   rec = thread->new_rec;
43   slot = new_slot();
44   slot->epoch = INDIRECT_EPOCH;
45   slot->prev = prev;
46   slot->next = next;
47   slot->pt = rec;
48   rec->heads[rec->count] = head;
49   rec->slots[rec->count++] = slot;
50   do {
51     old = head->slots;
52     slot->slot_next = old;
53   } while (!CAS(&(head->slots), old, slot));
54 }
```

Figure 9. A VLIST insert implementation.

4.2 List Writers

The primary differences between a write critical section and a read critical section is that a writer must log its changes, and, at the end of the critical section, either commit the changes or retry.

Figure 9 shows an implementation of the `insert()` and `delete()` functions using VLISTs. As with readers, the code has explicit critical section delimiters (`write_cs_enter()` and `write_cs_exit()`), and all `next` and `prev` pointer modifications use a helper function `add_slot`.

The helper function `add_slot()` serves two purposes. First, it creates a speculative slot for pending modifications. At Line 53, there is an atomic CAS to the head of the node-internal slot list, ensuring that all speculative updates are applied to the list node. The second purpose of this helper is to log this to the thread's `new_rec` update record, so that these updates can be either invalidated or committed at the end of the critical section.

The `write_cs_enter()` function is listed in Figure 8, and it is identical to `read_cs_enter` except that it also allocates a new update record for the thread.

The primary complication with writing is ensuring that updates are committed safely. Figure 8 includes code for `write_cs_exit()`. The first step in ending a write critical region is to detect conflicting updates. The loop beginning at Line 28 compares the set of list nodes with pending modifications to any updates since the critical section started. If any intervening update record wrote to any of the same nodes, the commit fails.

If a critical section fails, the speculative slots are marked as invalid (Line 48). If there are no conflicting updates, the writer attempts to add its own update record to the end of the log using an atomic CAS (Line 37). This is the linearization point of the algorithm. If the CAS fails, the thread continues walking forward on the list to search for conflicts, and then retries the CAS. If the CAS succeeds, the thread assigns the next epoch value to its update record.

After the CAS, the writer does some bookkeeping, primarily to remove now-obviated indirection and facilitate garbage collection. First, the writer stores the current epoch value in its committed update record. We also note that in Line 6, a thread may assist a committing writer that has linearized its update_record, but has not stored an epoch value. Because the epoch value is determined by the position in the list, multiple updates will be idempotent.

In order to optimize future list searches, the writer also propagates its epoch to all modified slots (Line 40). Note that, when a reader observes any slots with the reserved epoch value INDIRECT_EPOCH, it reads the epoch value from the update record (Figure 7, Line 15) If the update record's epoch is the reserved value `INACTIVE_EPOCH`, the slot is assumed speculative and ignored.

INVARIANT 4.2. *A thread may not begin a critical section with a given epoch value unless the epoch value is stored in an update record in the log.*

This invariant prevents threads from overlooking recently-committed slots, when executing concurrently with post-commit clean-up operations.

Programmers can write their own composed operations like `move` by guarding the critical section with existing `enter` and `exit` functions, calling `read_slot()` to traverse the list and using `add_slot()` to modify pointers.

Garbage Collection. As the global epoch clock advances, older slots and update records will not be read by threads with a higher epoch. Thus, the memory for obviated slots and update records must be reclaimed.

In order to reclaim an object safely, the algorithm must ensure that no threads are currently accessing the object. In order to detect whether a thread can be accessing a slot, VLIST adopts a quiescence-based scheme, similar to read-copy-update (RCU) [25]

At a high-level, quiescence-based memory reclamation first logically deletes items by placing them on a pending free list. In VLIST, when one writer commits a slot to a node, no future reader would access slots further down the slot list. Thus, these obviated slots can be added to a pending free list. For update records, old update records are also added to a pending free list once a new update record is committed.

Items on the pending free lists are physically freed once no threads can possibly access the data—generally leveraging some application-wide invariant. In the case of VLIST, we use the explicit critical section begin and end to identify when no threads could access to-be-freed slots. In other words, after each thread has exited a critical region, any slots on the pending free lists will never be accessed again. More formally, our quiescence library assures this invariant:

INVARIANT 4.3. *If an object o is added to a pending free list at time t, object o is only returned to the memory allocator if all threads have entered and exited a critical section* **after time t**.

4.3 Design and implementation issues

Shared State. All VLISTs in an application share a single epoch counter and list of committed update records. However, there is no single, shared pointer or integer which must be updated, reducing cache contention. The VLIST algorithm also relies on a quiescence-based memory reclamation library, which must detect when all threads have quiesced, commonly through a per-thread data structure.

Epoch Wrap-Around. Epochs in the prototype implementation are represented with 64-bit integers. There is a theoretical possibility, unlikely in practice, that a particularly old slot could persist across a wrap-around of the epoch counter. If an extremely efficient machine were able to process one trillion write critical sections per second (cf. current cores process at most 4 billion instructions/second), a wrap-around would occur after 5.8×10^{14} years.

Debugging. An interesting property of programming with a concurrent, versioned data structure is that concurrency errors are often easier to debug. In debugging the typical concurrency error on a traditional data structure, it is hard to reconstruct the state at the instant something went wrong. In VLISTs, the necessary state is likely to persist, as each list node stores a full history of versions. Moreover, a common failure mode of earlier versions of the VLIST implementation was for all threads to hang when trying to allocate a slot. This fail-stop mode combined with the old versions greatly simplified the task of reconstructing error behavior.

Application to other data structures. Any pointer-based data structure can be versioned by creating slots to track modifications. We also implemented versioned binary search trees and red-black trees. However, additional caution should be applied to balanced trees. During rebalancing, a writer can modify a node that it has modified before in the critical region. Thus, the writer should traverse its pending update record before allocating a new slot during rebalancing. Also, since writers need to maintain invariants of red-black tree, in some cases, speculative slots should be inserted to maintain color of nodes, even if the operation does not modify the nodes.

Integration with HTM. One way to view current best-effort HTM [1, 32] is providing the abstraction of large, but bounded, compare-and-swap. We expect that the VLIST design could easily leverage HTM to improve its performance, which we leave for future work. Specifically, we expect that adding speculative slots could be replaced with a hardware

transaction to add slots to each node atomically, along with an update record or a simple version counter. We expect the update log would have better coherence behavior, and the would eliminate post-commit bookkeeping.

5. Correctness

This section outlines correctness arguments for the VLIST algorithm, a full proof is omitted for space. To show correctness, we must establish the following:

1. Readers must see a consistent view of the data structure. Note that writers initially traverse the lists as readers to construct a set of pending updates.
   (a) Any bookkeeping to be garbage collected can only be (physically) freed when no thread can reference it.
2. A writer's updates are serializable.
3. At least one thread will always make progress (lock-freedom). Specifically:
   (a) All readers make progress.
   (b) At least one writer makes progress.

THEOREM 5.1. *Readers observe a consistent view of lists.*

When a reader begins a list traversal, it selects an epoch value (e) reflecting state committed to all lists. As each list node is traversed, any invalid, uncommitted, or newer slots (with epoch e' $>$ e) will be ignored (Figure 7, Lines 18–19). Because of Invariant 4.1, and the fact that `read_slot` stops at the first slot it observes with epoch value less than or equal to the thread's epoch (Figure 7, Line 19), The reader will observe a view of the list equivalent to the value at the point epoch e was committed.

LEMMA 5.2. *Slots are only garbage collected after all readers would read a more recent slot.*

A slot s is only added to the pending free list if a more recent slot s' exists. Future reader and writer threads will obtain the most recent epoch (e.g., Figure 7, Line 19), and will ignore this slot. Thus, only concurrent readers may need to read a slot on the pending free list. Invariant 4.3, provided by the garbage collection library, assures that pending slots will only be reclaimed after all concurrent readers have exited.

LEMMA 5.3. *If writer A begins at epoch e1, writer B begins at e2, both writers execute concurrently, and both writers add a speculative slot to node n, only one writer may commit at epoch e3 >e1,e2. The other will retry and may commit at epoch e4 >e3.*

This is assured by the conflict detection logic in `write_cs_exit()` (Figure 8, Lines 28), which checks all update records between the record associated with the beginning of a critical section and the end of the log. If any record includes a slot insert for the same node, a conflict is raised and the writer must retry.

THEOREM 5.4. *The updates applied to all lists by a writing thread are serializable.*

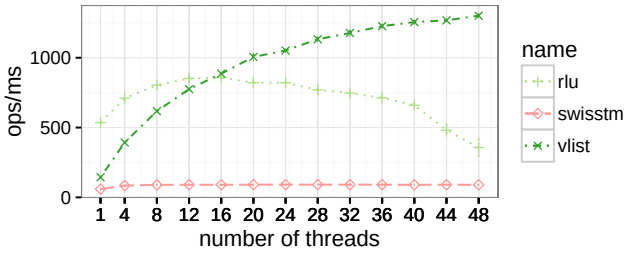

Figure 11. Move benchmark between two 1K-node list. Y-axis is operations per millisecond.

All potential writes are stored in speculative slots, which are atomically added to nodes (Figure 9, Line 53); thus, slot updates cannot be lost. Lemma 5.3 ensures that writes to a list node which would violate isolation must be retried. The fact that we are using a doubly-linked list ensures that checking the intersection of writes is sufficient to detect any violations of serializability, as any changes to a node also affects the previous and next neighbors—sufficient to detect any adjacent modification of the list structure.

Speculative slots are published to the rest of the system by a single, atomic CAS operation on the update record at the end of the log (Figure 8, Line 37), ensuring a total global ordering of updates, which is consistent with a sequential order. After this CAS succeeds, future critical sections will stop ignoring the newly added slots as speculative, as explained in §4.2.

THEOREM 5.5. *All readers will make progress.*

This follows from the fact that, at no point in the algorithm is a reader obstructed by another thread.

THEOREM 5.6. *At least one writer can always make progress.*

The only point at which writer A can retry part of the algorithm is if writer A detects an intervening, conflicting update record from writer B. Thus, during the execution of writer A's critical section, at least one writer (B) committed updates to the list.

Thus, VLISTs are lock-free. We note that, like any lock-free algorithm, a writer may starve (e.g., writer A above).

6. Evaluation

We evaluate the performance of versioned programming using linked-list and tree implementations in C. We compare to several competing approaches to implementing concurrent data structures. All measurements were taken on a 48-core SuperMicro SuperServer, with four 12-core AMD Opteron 6172 chips running at 2.1 GHz and 64 GB of RAM.

List benchmark details. We modify the benchmarking framework used by Read-Log-Update [24]; and compare to the Harris-Michael algorithm, RCU, RLU and SwissTM. For a fair comparison, we disable double-linking, which the other algorithms do not implement. The framework tests the

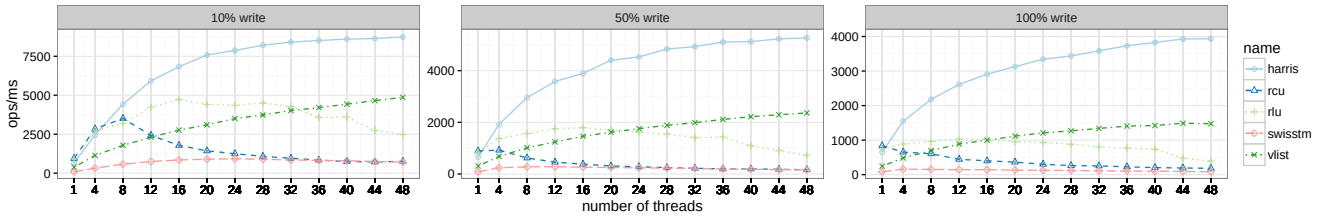

Figure 10. 10, 50 and 100 percent write microbenchmark on an 1K-node list. Y-axis is operations per millisecond.

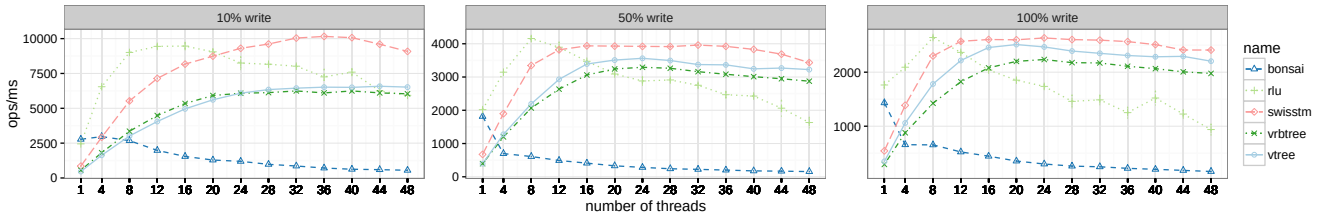

Figure 12. 10, 50 and 100 percent write microbenchmark on an 100K-node tree. Y-axis is operations per millisecond.

performance of a list implementation under mixes of read and write operations, and reports operations per millisecond.

We apply quiescent-state based reclamation (QSBR) as garbage collection for Harris-Michael algorithm. QSBR is cheaper than hazard pointers [27][13] used in the original RLU framework. The RCU implementation is from [2]. We disable RLU deferring [24], which can cause some updates to hang for a long time. SwissTM [10] (release 2011-08-05) is a high-quality STM with a epoch-based conflict detection mechanism similar to versioned programming.

We exercise each list implementation with a sorted list of 1000 initial nodes drawn from a 2000 integer range. The microbenchmark measures the total number of operations completed over roughly a one-second period and reports the average operations per millisecond and we run each setting five times. We vary the fraction of read versus write operations to exercise the major type of behavior each list would likely encounter.

List insert/remove performance. Figures 10 reports the performance of 10 percent write, 50 percent write and 100 percent write test cases. We selected a range of write levels to show that versioning can do well for read-mostly structures, but can also perform well with higher contention.

The Harris-Michael algorithm with QSBR exhibits the best overall performance for this benchmark. Unlike VLIST, Harris-Michael lists do not support composition of operations. Thus, each thread can commit updates independently, and there is only contentino when threads try to CAS the same next pointer.

RCU write performance is limited by the lock granularity. In this benchmark, we use a lock per list, which causes serialiation around the single lock. RLU performs better than RCU via finer-grained synchronization for writers. However, as number of threads increases, RLU quickly reaches a bottleneck to wait for a quiescent state to apply updates to lists.

VLIST eventually stops scaling at high thread counts. When there are enough rapidly-writing threads, every update is competing to update the update record list, which becomes a hot spot which incurs a lot of coherence misses. Nonetheless, the performance does not degrade under this level of contention, as is common in many lock implementations. Also, since there is another layer of indirection (from *node* to *slot*), single-threaded performance is often worse. We expect the single-threaded performance could be improved by compressing a "cold" slot into the main node when a node remains stable for a relatively long period.

SwissTM does even worse in the single-threaded scenario, and we observe many read validation failures with multiple threads.

List move performance. Note that the previous comparisons used only single inserts, deletions, or searches, on a singly-linked list, putting the VLIST algorithm at a relative disadvantage, as it incurs overheads to provide composability. Thus, Figure 11 shows an additional microbenchmark that measures the performance of moving elements between two 1000-element lists, protected by RLU, SwissTM and VLISTs. Other lock-free algorithms cannot safely execute this microbenchmark.

As expected, the VLIST algorithm substantially outperforms RLU at 16 threads and higher, and has strictly better performance scalability than SwissTM. This improvement is attributable to VLIST's ability to avoid conflicts on list elements during the search phase, and only detect conflicts while moving elements.

Tree insert/remove performance. We use versioned programming to implement a binary search tree VTREE and a red-black tree VRBTREE, which we compare to a SwissTM red-black tree, RCU-based self-balancing bonsai tree [7] and a binary search tree described in RLU [24]. The benchmark

does read and write to a tree initialized with 100K nodes. Note, although VTREE and RLU tree do not rebalance, the penalty is small because the insertions are random.

Figure 12 shows the tree performance. With higher writer counts, VTREE improves performance over RLU, commensurate with previous results. Bonsai trees are relatively expensive, as they must build a new tree for each write operation, which requires a global lock. SwissTM performs well on the tree benchmark, as contention is relatively low; the VRBTREE curve has a similar shape, but higher overall costs.

7. Related Work

Valois [36] is credited with the first practical lock-free design for a singly-linked list. Harris [15] describes a simpler approach which was further refined by Michael [26], making it compatible with several lock-free memory management methods, including hazard pointers [27]. Fomitchev and Ruppert [12] further refine the worst-case amortized cost of operations. Sundell and Tsigas [35] approximate double-linking by marking previous pointers unreliable.

Liu et al. [22] explore bounded staleness and a versioned consensus protocol (similar in some respects to versioned programming), to ensure scalable performance in read-mostly reader-writer locks (prwlock). Prwlock can be seen as an improvement to RCU/RLU, and will probably yield the most benefit for read-mostly data structures, whereas versioned programming improves performance with modest-to-high write ratios.

Universal constructions. The earliest algorithms for lock-free linked lists were Herlihy's universal constructions [17, 18]. Universal constructions can turn any sequential algorithm into a parallel algorithm with certain properties, such as wait-freedom [6]. These constructions are useful for reasoning about asymptotic behavior and writing proofs, but generality often comes at a substantial performance cost and these are rarely used in deployed software.

Dang and Tsigas [8] demonstrate that some compositions of lock-free data structures can violate lock-freedom.

DCAS-based algorithms. The Motorola 68k architecture provided a double word compare-and-swap (DCAS) instruction, but has performance and implementation issues that have prevented adoption on any ISA since. Several research operating systems in the early 90s used this instruction to implement lock-free linked lists [14, 23]. Cederman and Tsigas [5] implemented a lock-free move by combining remove and insert in a single DCAS operation. Best-effort hardware transactional memory might revive these algorithms, as it it becomes part of commodity systems [1]. In order to be widely deployed on commodity systems, algorithms generally rely on a single-word CAS.

Software Transactional Memory (STM) provides atomic and isolated updates to arbitrary memory locations [9, 10]. While some STM implementations are *obstruction-free* [20], most use two-phase locking and simply encapsulate the locking complexity [9, 10].

Several STM implementations, including TL2 [9], TinySTM [11], SwissTM [10], and others [37], use a global version clock to detect conflicts. A transaction records the clock value when it begins, and if it sees more recent versions, it aborts. Several papers have developed sophisticated (and orthogonal) techniques to avoid needless rollback by extending the range of acceptable timestamp values, with additional bookkeeping [34] and relaxed consistency [33]. In some respects, VLISTs can be seen as a specialization and extension of these global clock techniques. VLISTs innovate in their degree of progress guarantees—to our knowledge, no STM based on a global version clock is also lock-free. STMs generally keep only one shared, visible version of each object; readers that observe an old version must abort and restart. In most cases, reader progress is not guaranteed unless they "upgrade" to write mode. SwissTM and its predecessors keep multiple old versions to ameliorate this issue, but cannot guarantee a requested version will be available. In contrast, VLISTs ensure that old versions are not overwritten until all potential readers have completed.

JVSTM [4] keeps the entire history for an object, so readers always have a consistent view of object versions. JVSTM uses dynamic analysis in the JVM to garbage collect inaccessible versions. Although JVSTM provides the programmer with guarantees closest to VLISTs, versioned programming has lighter runtime requirements.

Concurrent Trees. Lock-free trees are hard to write because delete operations can race with deletion (similar to lists in §2), as well as with rebalancing the tree. Most concurrent trees relax balancing constraints and use locking to serialize writers, but avoid locking readers [2, 7, 24].

8. Conclusion

Lock-free data structures are hard to implement, and this paper presents a practical toolkit to turn sequential implementations into lock-free implementations. The API for versioning is simple, as is the implementation, making it suitable for use as a drop-in replacement for popular concurrency packages. This work improves performance over other, lock-based frameworks under modest write contention or thread counts, and can expand the reach of data structures that can be made lock-free. Our code is available at `https://github.com/oscarlab/versioning`.

Acknowledgments

We thank the anonymous reviewers for insightful comments on this work. Chia-Che Tsai, William Jannen, and Oliver Jensen contributed to the prototype. This research was supported in part by NSF grants CNS-1149229, CNS-1161541, CNS-1228839, CNS-1405641, CNS-1408695, CNS-1526707, VMware, and the Office of the Vice President for Research at Stony Brook University.

References

[1] AMD. Advanced synchronization facility proposed architectural specification. `http://developer.amd.com/assets/45432-ASF_Spec_2.1.pdf`, March 2009.

[2] M. Arbel and H. Attiya. Concurrent updates with rcu: Search tree as an example. pages 196–205. ACM, 2014.

[3] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[4] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005.

[5] D. Cederman and P. Tsigas. Supporting lock-free composition of concurrent data objects: Moving data between containers. *Computers, IEEE Transactions on (TC)*, 62(9):1866–1878, Sept 2013.

[6] P. Chuong, F. Ellen, and V. Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 335–344, 2010.

[7] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using **rcu** balanced trees. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[8] N. N. Dang and P. Tsigas. Progress guarantees when composing lock-free objects. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 148–159, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the International Conference on Distributed Computing (DISC)*, pages 194–208, 2006.

[10] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 155–165, 2009.

[11] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 237–246, 2008.

[12] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the ACM symposium on Principles of distributed computing (PODC)*, pages 50–59, 2004.

[13] K. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge Computer Laboratory, 2004.

[14] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 123–136, 1996.

[15] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the International Conference on Distributed Computing (DISC)*, pages 300–314. Springer-Verlag, 2001.

[16] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, 2007.

[17] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[18] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.

[19] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *PPoPP*, 2008.

[20] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the ACM symposium on Principles of distributed computing (PODC)*, 2003.

[21] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[22] R. Liu, H. Zhang, and H. Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *Proceedings of the USENIX Annual Technical Conference*, pages 219–230, Philadelphia, PA, June 2014. USENIX Association.

[23] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. *SIGOPS Oper. Syst. Rev.*, 26(2):8, 1992.

[24] A. Matveev, N. Shavit, P. Felber, and P. Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2015.

[25] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy Update Techniques in Operating System Kernels*. PhD thesis, Oregon Health and Science University, 2004.

[26] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 73–82, 2002.

[27] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[28] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[29] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 68–78, 2007.

[30] E. Petrank, M. Musuvathi, and B. Steensgaard. Progress guarantee for parallel programs via bounded lock-freedom. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 144–154, 2009.

[31] D. P. Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, Massachusetts Institute of Technology, 1978.

[32] J. Reinders. Transactional synchronization in haswell. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell`, 2012.

[33] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the International Conference on Distributed Computing (DISC)*, pages 284–298, 2006.

[34] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *Proceedings of the ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2006.

[35] H. Sundell and P. Tsigas. Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, 2005.

[36] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the ACM symposium on Principles of distributed computing (PODC)*, pages 214–222, New York, NY, USA, 1995. ACM.

[37] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 34–48, 2007.