Multiversion Concurrency with Bounded Delay and Precise Garbage Collection

Naama Ben-David  
Carnegie Mellon University  
nbendavi@cs.cmu.edu

Guy E. Blelloch  
Carnegie Mellon University  
guyb@cs.cmu.edu

Yihan Sun  
Carnegie Mellon University  
yihans@cs.cmu.edu

Yuanhao Wei  
Carnegie Mellon University  
yuanhao1@cs.cmu.edu

ABSTRACT

In this paper we are interested in bounding the number of instructions taken to process transactions. The main result is a multiversion transactional system that supports constant delay (extra instructions beyond running in isolation) for all read-only transactions, delay equal to the number of processes for writing transactions that are not concurrent with other writers, and lock-freedom for concurrent writers. The system supports precise garbage collection in that versions are identified for collection as soon as the last transaction releases them. As far as we know these are first results that bound delays for multiple readers and even a single writer. The approach is particularly useful in situations where read-transactions dominate write transactions, or where write transactions come in as streams or batches and can be processed by a single writer (possibly in parallel).

The approach is based on using functional data structures to support multiple versions, and an efficient solution to the Version Maintenance (VM) problem for acquiring, updating and releasing versions. Our solution to the VM problem is precise, safe and wait free (PSWF).

We experimentally validate our approach by applying it to balanced tree data structure for maintaining ordered maps. We test the transactional system using multiple algorithms for the VM problem, including our PSWF VM algorithm, and implementations with weaker guarantees based on epochs, hazard pointers, and read-copy-update. To evaluate the functional data structure for concurrency and multi-versioning, we implement batched updates for functional tree structures and compare the performance with state-of-the-art concurrent data structures for balanced trees. The experiments indicate our approach works well in practice over a broad set of criteria.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPAA ’19, June 22–24, 2019, Phoenix, AZ, USA  
© 2019 Association for Computing Machinery.  
ACM ISBN 978-1-4503-6184-2/19/06 . . . $15.00  
https://doi.org/10.1145/3323165.3323185

1 INTRODUCTION

Consider a sequential computation that takes $\tau$ instructions (time) to run. If the computation is run by some system atomically as a transaction concurrently with other transactions that share data, we would expect it would take more time to complete. This can be both due to the overhead of the transactional system, and due to inherent dependences among the transactions, forcing the system to wait for another to complete. In this paper we are interested in bounding the extra time. We say the sequential computation has $O(\delta)$ delay if its transaction completes in $O(\tau + \delta)$ time.

In general, it is impossible to bound the delay by better than $O(\tau \times p)$, even ignoring overheads. Several approaches try to take advantage of this. Copy-read-update (RCU) [44] allows for an arbitrary number of readers to proceed with constant delay, and has become a core idiom widely used in Linux and other operating systems [43]. In RCU, however, readers can arbitrarily delay (block) a writer, since a writer cannot proceed until all readers have exited their transaction. This is particularly problematic if some readers take significant time, fault, or sleep [41]. Indeed RCU in Linux is used in a context in which the readers are short and cannot be interrupted.

Multi-versioning allows for an arbitrary number of readers to proceed with constant delay, but in principle, they can avoid delaying any writers—a writer can update a new version while readers continue working on old versions. Therefore a single writer and any number of readers should all be able to proceed without delay (multiple writers can still delay each other).

Multi-versioning, however, has some significant implementation issues that can make the “in principle” difficult to achieve in “theory” or “practice”. One is that memory can become an issue due to maintaining old versions, possibly leading to unbounded memory usage. Ideally one would like to reclaim the memory used by a version as soon as the last transaction using it finishes. Some recent work has studied such bounds in memory usage [52]. Although

1Throughout we use “transaction” to mean the traditional sense of a sequence of instructions that appear to take place atomically at some point during their execution (strictly serializable) [50], and not to mean a specific implementation technique such as transactional memory.
their results ensure readers are not blocked and do not block writers, they do not bound delay. Another problem arises in the most widely used implementation of multi-versioning, which involves keeping a version list for every object [13, 39, 51, 56]. The problem is that these lists need to be traversed to find the relevant version, which causes extra delay for reads. The delay is not just a constant, but can be asymptotic in the number of versions. We know of no multi-versioned system that can both bound the delay and ensure memory usage bounds, even when only a single writer is allowed at any time.

In this paper, we develop strong asymptotic bounds on the delay for transactions while also ensuring bounded memory. We show what we believe are the first non-trivial cost bounds for transactions with multi-versioning. In particular, for processes we describe a system with the following properties:

- Read transaction are delay-free—i.e., if they take time (instructions) in the original code, they take time in the transactional version, from invocation to response.
- A single write transaction (without other concurrent write transactions) has delay from invocation to response (i.e., when the result is visible).
- Multiple concurrent write transactions are lock-free, although a successful write will abort other active writers.
- The garbage collector is precise in that the memory associated with any version (except the latest) is collected as soon as the last transaction that holds it completes. Furthermore, the cost of the collection is linear in the amount of garbage collected.
- A single writer transaction along with read transactions (not including the garbage collection) have constant amortized memory contention.

These properties are true for arbitrarily long transactions that access an arbitrary memory footprint for read-only transactions, and update an arbitrary number of locations for writing transactions.

Our approach is particularly useful in read-dominated workloads in which a single (or very few) writer does updates, or in workloads in which concurrent writes can be batched into single transactions in the style of flat-combining [30], and then applied by a single writer. As with flat-combining, batching gives up on the wait-freedom of writes, however it allows the writes to run in parallel potentially getting high throughput. We study this in our experiments.

To achieve these bounds we require that programs are implemented using purely functional data structures [8, 38, 47, 53]. Such data structures are widely used in languages such as F#, Scala, OCaml, Haskell, JavaScript, Julia, and Clojure, and date back to the 1950s with Lisp [42]. They are also used in various database systems [1, 4, 14, 28], and sometimes referred to as copy-on-write [7, 58]. On updates, the path to the update is copied. Most standard data types can be implemented efficiently (asymptotically) in the functional setting, including balanced trees, queues, stacks and priority queues. Since functional data structures are persistent (immutable), they are naturally multi-versioned. Applying an update leaves the old version while creating a new version. The version can be accessed via a pointer to the root, and hence each version is simply a pointer to a data structure. The cost of traversing the structures is unaffected by the versions (unlike version lists). However, the problem remains of how to ensure precise garbage collection.

For the purpose of garbage collection, we introduce the version maintenance (VM) problem. The problem is to implement a linearizable object with three operations: acquire, release and set. The acquire operation returns a handle to the most recent version, in a way that ensures it cannot be collected. The set operation updates the current version to a new pointer, returning whether it succeeded or failed. The release operation indicates that the currently acquired version is no longer needed by the process, potentially making it available to be collected. It returns a list of versions that can be collected—i.e., for which no other process has acquired it and not released it. Only one version can be acquired on any process at any time, i.e. the current version must be released before a new one is acquired. In the precise VM problem, the release will return a singleton list precisely when the process is the last to release its version, and an empty list otherwise. We give a solution to the precise version.

The VM object can be used to implement read-only and writing transactions as shown in Figure 1. The read transaction is effectively done after step 2 (response could be sent to a client), and the rest is a cleanup phase for the purpose of GC. Similarly, writing transactions are done after step 3, at which point the result is visible to other transactions. After the release, any garbage can be traced from the released pointers and collected in work linear in the amount of garbage collected using a standard reference counting collector.

We describe a wait-free algorithm for the precise VM problem, which we refer to as the PSWF algorithm. It supports the acquire
with \(O(1)\) delay, and set and release with \(O(p)\) delay. A read-only transaction only costs the delay of an acquire (constant), followed by the cost of the transaction itself, which is unaffected by the multi-versioning (e.g., a search in a balanced tree will take \(O(\log n)\) time). In our implementation, the set can only fail if a concurrent writer has succeeded between its acquire and set. Therefore a non-conflict writing transaction takes effect in the time of the transaction itself plus the cost of the acquire and set, which is \(O(p)\) time (for the set). We also consider the memory contention of the three operations. The costs and properties are summarized in Table 1.

We finish by describing some experiments for both the VM algorithms and the functional data structures. We test the transactional system using multiple VM algorithms in our framework, including our PSWF algorithm, and implementations with weaker guarantees based on epochs and hazard pointers. Experiments show that our PSWF algorithm on average uses 60%-90% less memory for versions than the other two implementations because of precise garbage collection. Our algorithm also achieves comparable throughput to the other two implementations.

To evaluate the functional data structure for concurrency and multi-versioning, we implement batched updates for functional trees and compare the performance with existing concurrent data structures. Experiments show that in the tested workloads with mixed reads and updates, using functional data structures with batching can outperform concurrent data structures by more than 20%.

## 2 Preliminaries

We consider asynchronous shared memory with \(P\) processes. Each process \(p\) follows a deterministic sequential protocol composed of primitive operations (read, write, or compare-and-swap) to implement an object. We define objects, operations and histories in the standard way [34]. We consider linearizability as our correctness criterion [32, 35]. An adversarial scheduler determines the order of the invocations and responses in a history. We refer to some point in a history as a configuration. We define the time complexity of an operation to be the number of instructions (both local and shared) that it performs. Note that this is different from the standard notion of step complexity which only counts access to shared variables.

**Transactions.** We consider two types of transactions: read-only and write. Each transaction has an invocation, a response, and a completion, in that order. A transaction is considered active between its invocation and response, and live between its invocation and completion. Intuitively, the transaction is executed between its invocation and response, and does some extra ‘clean-up’ between its response and completion. We require that transactions be strictly serializable, meaning that each transaction appears to take effect at some point during its active interval. We refer to a write transaction as single-writer if no other write transaction is live while it is live.

**Delay.** We say that the time of a computation (or algorithm) on a single process is the number of instruction steps that the computation executes, including all local and shared instructions. We say that the user instructions of a transaction are the instructions that would be run in a sequential setting using regular reads and writes. We want to simulate these instructions in a way that the transaction appears atomically in the concurrent setting. Consider a transaction that executes user code that consists of \(m\) user instructions. Such

![Figure 2: An example of the insert function under PLM using path copying.](image)

a simulation has delay \(d\) if the active interval takes \(O(d + m)\) time, similarly to [9]. A transaction is delay-free if the delay is constant (or zero). The \(O(d + m)\) bound includes all instructions needed to ensure strict serializability, and the big-O is independent of the number of processes, the number of versions, or the actions of any other concurrent processes. In a traditional multiversion system, for example, the bound needs to include the possibly large number of instructions needed to traverse a version list.

**Contention.** We say that the amount of contention experienced by a single shared-memory operation \(i\) in a history \(H\) is the number of responses to modifying operations on the same location that occur between \(i\)’s invocation and response in \(H\). Note that this is not exactly the definition presented in any previous paper, but it is strictly stronger (implies more contention) than both the definition of Ben-David and Blelloch [10] and the definition of Fich et al. [26]. Therefore, the contention results in this paper hold under the other models as well.

**Functional Data Structures.** We assume that the memory shared by transactions is based on purely functional (mutation-free) data structures. This can be abstracted as the pure LISP machine [8, 47, 53] (PLM), which, like the random access machine model (RAM), has some constant number of registers. However, the only instructions for manipulating memory, are (1) a tuple\((v_1, \ldots, v_k)\) instruction, which takes \(l\) registers (for some small constant \(l\)) and creates a tuple in memory containing their values, and (2) a \(\text{nth}(t, i)\) instruction, which, given a pointer \(t\) to a tuple and an integer \(i\) (both in registers), returns the \(i\)-th element in this tuple. Values in the registers and tuples are either primitive, or a pointer to another tuple. There is no instruction for modifying a tuple. Changing a data structure using PLM instructions are done via path copying, meaning that to change a node, its ancestors in the data structure must be copied into new tuples, but the remainder of the data remains untouched. Using PLM instructions, one can create a DAG in memory, which we refer to as the memory graph. A special and commonly-used case for the memory graph is a tree structure.

We define the version root as a pointer to a tuple, such that the data reachable from this tuple constitutes the state that is visible to a transaction. Then each update on version \(v\) yields a new version by path-copying starting from the version root of \(v\), and the new copied root provides the view to the new version. An example of using path-copying to insert a value into a binary tree memory graph is shown in Figure 2. In our framework, every transaction \(t\) acquires exactly one version \(V(t)\). If \(t\) has not yet determined its version at configuration \(C\), then \(V_C(t) = \text{null}\) until it does. We use the version roots as the data pointers in the Version Maintenance problem.

**Garbage Collection.** We assume all tuples are allocated at their tuple instruction, and freed by a free instruction in the GC. The
allocated space consists of all tuples that are allocated and not yet freed. For a set of transactions \( T \), let \( R(T) \), or the reachable space for \( T \) in configuration \( C \), be the set of tuples that are reachable in the memory graph from their corresponding version roots, plus the current version \( c \), i.e., the tuples reachable from \( \{ V(t) \mid t \in T \} \cup \{ c \} \). We say that a tuple \( u \) belongs to a version \( v \) if \( u \) is reachable from \( v \)'s version root. Note that \( u \) can belong to multiple versions. We define a precise and a safe GC, respectively, as follows.

**Definition 2.1.** A garbage collection is precise if the allocated space at any point in the user history is a subset of the reachable space \( R(T) \) from the set of live transactions \( T \).

**Definition 2.2.** A garbage collection is safe if the allocated space is always a superset of the reachable space from the active transactions.

Roughly speaking, precise GC means to free any out-of-date tuples in time, and safe GC means not to free any tuples that are currently used by a transaction.

### 3 THE VERSION MAINTENANCE PROBLEM

In our transaction framework, we abstract what we need for the purpose of maintaining versions as the **Version Maintenance** problem, which tackles entering and exiting the transactions (see Figure 1).

The Version Maintenance problem, or Version Maintenance object, supports three operations: \texttt{set}, \texttt{acquire}, and \texttt{release}. At a high level, the \texttt{acquire} operation returns a version for the process to use and \texttt{release} is called when the process finishes using the version. New versions are created by \texttt{set} operations. All three operations take as input an integer \( k \) that represents the id of the process that calls the operation. The \texttt{set} operation in addition takes in a pointer to the new version that it should commit, and returns a flag indicating whether or not it succeeded.

We refer to the pointer to a version as the **data pointer**. More formally, if \( d \) is a pointer to data, \texttt{set}(\( d \)), if successful, creates a new version with pointer \( d \) and sets it as the **current version**, i.e.,

**Definition 3.1.** The current version is defined as the version set by the most recent successful \texttt{set} operation.

The operations are intended to be used in a specific order: an \texttt{acquire} operation should be followed by a \texttt{release} operation, with at most one \texttt{set}, \( d \), in between, where \( d \) is a pointer to a new version. If this order is not followed for each \( k \), then the operations may behave arbitrarily; that is, we do not specify a ‘correct’ behavior for the operations of a Version Maintenance object \( O \) in an execution once any operations are called out of this order on \( O \).

We define the liveness of a version \( v \) as follows.

**Definition 3.2.** A version \( v \) is live at time \( t \) if it is the current version at \( t \), or if \( \exists k, s.t. \text{ an } \texttt{acquire} \text{ operation } A \text{ has returned } v \text{ but no } \texttt{release} \text{ has completed after } A \text{ and before } t \).

We note that a version is live while a transaction using that version is active. The transaction itself can remain live after its version is dead, while it garbage collects.

The following is the sequential specification of these operations assuming that they are called in the correct order (\texttt{acquire-release} or \texttt{acquire-set-release} for each id \( k \)).

- \texttt{data* acquire(int k)}: Returns the current version.
- \texttt{data** release(int k)}: Returns a (possibly empty) list of versions that are no longer live. No version can be returned by two separate \texttt{release} operations.
- \texttt{bool set(int k, data* d)}: Sets the version pointed to by \( d \) as the current version. Returns true if successful. May also return false if there has been a successful \texttt{set} between this \texttt{set} and the most recent \texttt{acquire} of \( k \). If the \texttt{set} returns false, it has no effect on the state of the object.

We say that a process \( p \) has acquired version \( v \) if \( \text{acquire}(k) \) returns \( v \), and say \( p \) has released \( v \) when the next \texttt{release}(\( k \)) operation returns. If a \texttt{set} operation returns true, we say that it was **successful**. Otherwise, we say that the \texttt{set} was **unsuccessful** or that the \texttt{set} **aborted**. Note that conditions for correct aborting for the \texttt{set} are reminiscent of 1-abortability defined by Ben-David et al. [12], but we relax the requirements to allow a successful \texttt{set} to cause other \texttt{sets} to abort even if it was not directly concurrent with them, but happened sometime since that process’s last \texttt{acquire}.

An implementation of a Version Maintenance object is considered **correct** if it is linearizable as long as no two operations with the same input \( k \) run concurrently. Furthermore, it is considered **precise** if the \texttt{release} operation returns exactly the versions that stop being live at the moment the \texttt{release} operation returns. Note that this means that in a precise implementation of the Version Maintenance problem, each \texttt{release} operation \( r \) returns a list containing at most one version, and this version must be the one that \( r \) released. We show some properties of a correct Version Maintenance in the full version of this paper.

Where convenient, for a version \( v \), we use \texttt{acquire}_\( v \), \texttt{release}_\( v \), and \texttt{set}_\( v \) to denote an \texttt{acquire} operation that acquires \( v \), a \texttt{release} operation that releases \( v \), and a \texttt{set} operation that sets \( v \) as the current version, respectively.

#### 3.1 The PSWF Algorithm

We now present a simple wait-free algorithm that solves the precise version maintenance problem. That is, the \texttt{release} operation returns either an empty list of versions, or a singleton containing the version that it is releasing. We show that our wait-free algorithm is linearizable, and analyze it to obtain strong time complexity bounds; the \texttt{acquire} operation takes \( O(1) \) time, and the \texttt{release} and \texttt{set} operations each take \( O(P) \) time. Furthermore, we show that in the single-writer setting, where concurrent \texttt{set} operations are disallowed, the algorithm guarantees amortized constant contention per shared-memory operation. These properties show that regardless of adversarial scheduling, version maintenance need not be a bottleneck for transactions. The main results are shown in Theorem 3.3, 3.4 and 3.5. All proofs are in the full version. Pseudocode for the algorithm is given in Algorithm 4, and Figure 3 shows how its data is organized.

To understand the idea behind our algorithm, consider the following simplified (but incorrect) implementation. To set a new version, a process \( p \) simply \texttt{CASes} its data pointer into a global **currentVersion** location. If its \texttt{CAS} fails then it aborts. To acquire a version, \( p \) reads the currentVersion and copies it over to \( p \)'s slot in an **AnnouncementArray**, thereby signaling to others that it is using this version. The acquire operation then returns the version that it read.
When releasing a version \( v \), \( p \) scans the AnnouncementArray to see whether anyone else is still using \( v \). If not, \( p \) returns \( v \), as it is the last process that used this version. Otherwise, \( p \)'s release returns an empty list. This simple outline of an algorithm for the precise Version Maintenance problem satisfies the intuition of what should happen in a solution to the Version Maintenance problem; processes always acquire the current version, and return a version from their release operation only if this version stops being live at the end of the operation. However, this algorithm does not work in a completely asynchronous setting.

To see why, first note that a process \( p \) that executes an acquire operation may stall for a long time after reading the currentVersion but before announcing what it read. This could lead to a situation in which, by the time \( p \) announces the version \( v \) that it read, \( v \) has long since stopped being live, and has already been returned by some release operation. This scenario is not linearizable. We must also ensure that exactly one releasing process returns each version, meaning that an order between concurrent releasers must be established. Finally, we need to ensure that if a set aborts, then it or its preceding acquire were concurrent with a successful set.

To fix the acquire operation, we assign each process a 'helping' flag in its announcement slot, and use that flag to create two stages of the acquire operation; first a version is read from the current version field, \( V \), and announced with a 'helping' flag set, meaning that this is the version that the process intends to use, but has not started accessing yet. To secure this version, the acquiring process, \( p \), must reread the current version to ensure that it has not changed, and then set the 'helping' flag to false. In the meantime, other processes may see \( p \)'s announcement, and help it complete its acquire. Some set operations will try to help the acquires, so that no acquire can repeatedly fail without receiving help. Once the flag is down, \( p \) is said to have committed its announced version. In this way, the releasing process returning the version \( v \) can ensure that no process can acquire (commit) the same version \( v \) after it terminates.

To ensure that each version is only ever return by one release operation, we assign each version a "status" (stored in the array \( S \)), which can be in one of three states at any given time: usable, pending, and frozen. A release operation mainly deals with two things: helping all other processes complete their acquire on version \( v \), when necessary, and deciding if this is the last usage of version \( v \), and returning true if so. If \( v \) is usable, it means that no release operation is currently in progress on \( v \), and \( v \) may be in use. If a releasing process \( p \) sees this status, it tries to switch its status to pending, and if it succeeds, then it starts scanning the announcement array. While \( v \) is pending, a single releasing process is scanning the announcement array, and helping any process that has announced \( v \) to complete its acquire. Any releasing process that observes that \( v \) is already in the pending state can safely return false because there are currently other processes releasing this version. Once \( p \) has done scanning the array, it sets \( v \)'s status to frozen. This indicates to all other releasing processes that \( v \) if no process currently has \( p \) acquired, then \( v \) can never again be acquired by any new process. Thus, if no process currently has \( v \) announced, it is safe to return true on a release of \( v \). To ensure that only one releaser does so, the releasers of \( v \) compete in erasing \( v \) from the status array, and only the winner returns true.

Finally, we allow the set operation by process \( p \) to abort only under two conditions: (1) the current version \( V \) is not the same as \( p \)'s acquired version (in this case, it is easy to see that there must have been a successful set operation since \( p \)'s acquire); or (2) the set operation cannot find a spot in which to place its new version. That is, we have an array called \( S \) of versions that are currently active, and it is preallocated with a specific number of slots. Each set operation scans the array of versions to try to find an empty slot in which it can place its new version. The intuition is that if it cannot find an empty slot, then there must have been many other set operations concurrent with it. By setting the size of \( S \) to be large enough (\( 3P + 1 \) in our case), we can ensure that if a set operation \( op \) does not find any empty slots, there must have been some process that has executed a successful set during \( op \)'s interval.

We now describe the algorithm in more detail. A version \( v \) is represented as a pair of a timestamp and an index. If \( v \) is alive, the status of \( v \) is stored in \( S[v.index] \) (the Status array) and its associated data pointer is stored in \( D[v.index] \) (the VersionData array). For the rest of the paper, when we refer to a version, we mean a timestamp-index pair. Since there are at most \( P + 1 \) live versions, and at most \( P \) active set operations that could occupy another slot with a potential version, the Status and Data arrays can never have more than \( 2P + 1 \) occupied slots. However, for the purpose of guaranteeing that a set operation will only abort if it was concurrent with a successful set, we initialize \( S \) and \( D \) to be of size \( 3P + 1 \). Each slot \( A[k] \) in the announcement array belongs to process \( p_k \), and stores a help flag and a version. A global variable \( V \) stores the current version.

Set. To execute a set(d) operation for a data pointer \( d \), a process \( p \) first creates a new version \( v \) locally, and then looks for an empty slot for \( v \) in the status array. If it does not find an empty slot, then it aborts. Intuitively, it is ok to abort at that stage because at any given moment, \( S \) can have at most \( 2P \) occupied slots (one version acquired by each process, and another version that is in the middle of being set by each process). So, if \( p \) finds all \( 3P + 1 \) slots occupied,
it means that it was concurrent with \(2P + 1\) other set operations. Since there are only \(P\) processes, at least one process \(q\) executed 3 set operations concurrently with \(p\)'s set. If one of \(q\)'s sets were successful, \(p\) can safely abort its own operation. Otherwise, all 3 of \(q\)'s operations must have been concurrent with a successful set (for \(q\) to legally abort), and therefore, at least one of those successful sets must have been concurrent with \(p\)'s.

Now we assume that \(p\) did find an empty slot in \(S\). Let \(i\) be the index of this empty slot. \(p\) initializes \(S[i]\) with the new version, and writes \(d\) into \(D[i]\). Before setting \(v\) as the current version and terminating, \(p\) scans the announcement array, and helps every process that needs help (i.e. \(A[k] = (\text{true, } +)\)). To ensure that the helping is successful, \(p\) needs to perform three CAS operations on \(A[k]\). Each CAS tries to set \(A[k]\) to \((0, \text{oldVer})\), where oldVer is the version that \(p\) currently has acquired (announced in \(A[p]\)). To ensure that oldVer is still valid, \(p\) checks whether it is still the current version. If it is not, \(p\) aborts. These CAS operations can be thwarted at most twice by the acquire\((k)\) that requested help, so that the help is guaranteed to have succeeded after the third CAS. Finally, \(p\) tries to set \(v\) as the current version by CASing it into \(v\). If this CAS succeeds, so does \(p\)'s set operation. If it fails, \(p\) aborts, and first clears the slot it occupied in \(S\) to allow others to use it.

**Acquire.** The acquire\((k)\) operation begins by requesting help, reading the current version \(v\), and announcing it in \(A[k]\). To ensure that \(v\) is still the current version at the announcing step, the operation reads \(v\) again. There are two cases. If it finds that \(v\) has been updated, it starts over. It will only ever restart once, because if it finds that \(v\) has been updated once again, it knows that two set operations have occurred, one of which must have committed a version into \(A[k]\) by performing 3 helping CASes. If \(v\) is still the current version, we use a CAS to set the helping flag in \(A[k]\) to 0. Even if this CAS fails, \(A[k]\)'s helping flag must now be 0, since an acquire\((k)\)'s CAS only fails if it was helped by another process (a set or a release operation). Once acquire\((k)\) successfully commits a version \(v\), it reads and returns the corresponding data pointer \(D[v.index]\).

**Release.** To perform a release\((k)\) operation, the process \(p_k\) first reads the committed version \(v\) from its announcement slot, and clears the slot. If \(v\) is still current, the release\((k)\) operation returns false because \(v\) is still live. Otherwise, it must check whether someone else is still using \(v\). This is done by looking at the status at \(S[v.index]\). \(S[v.index]\) might be empty or store a version other than \(v\). In that case, some other release\((v)\) has already returned true, so \(p_k\) returns false. Otherwise, if \(S[v.index]\) stores a valid status (usable, pending, or frozen), then \(p_k\) uses this status to determine what to do, as described earlier.

This algorithm can be shown to be correct (linearizable) and efficient. We summarize the results as follows:

**Theorem 3.3 (Correctness).** Algorithm 4 is a linearizable solution to the Version Maintenance Problem.

**Theorem 3.4 (Step Bounds).** Each acquire\() operation requires at most \(O(1)\) time and each release() and set() operation requires \(O(P)\) time.

**Theorem 3.5 (Amortized Contention).** When concurrent set operations are disallowed, each acquire() operation experiences \(O(1)\) amortized contention and each release() and set() operation experiences \(O(P)\) amortized contention. Furthermore, no contention experienced by acquire() is amortized to release() or set().

Due to lack of space, we show the proofs in the full version. We note that Theorem 3.5 shows a property that is non-trivial to be achieved in wait-free algorithms, even in the single-writer setting—regardless of the adversarial scheduler, processes do not often contend on the same operations. Intuitively, our algorithm achieves this because of the version status: instead of allowing many releasing processes to traverse and modify the announcement array for every version, only one process per version (the one that changed the status from usable to pending) can do this at any given time. Furthermore, each slot in the announcement array can only have one version associated with it at any given time, meaning that only one releaser, one acquirer, and one setter can contend on any given slot.

### 4 Garbage Collection

In this section, we show how efficiently collect out-of-date tuples on functional data structures in the context of transactions and the VM problem. We first define the desired properties of GC on functional data structures. We then present the collect algorithm for our transactions (Figure 1) and show that it is fast and correct.

Intuitively, a linearizable precise VM solution provides an interface for safe and precise garbage collection over versions, since release\(_p\) returns true if and only if it is the last usage of \(v\). However, the precision and safety on the granularity of tuples relies on a “correct” collect operation, which, intuitively, should free all tuples that are no longer reachable as soon as possible. We formally define the desired property of a correct collect operation.

**Definition 4.1.** Let \(u\) be a tuple, and \(t\) be any time during an execution. A collect is correct if the following conditions hold.

- If for each version \(v\) that \(u\) belongs to, collect\((v)\) has terminated by time \(t\), then \(u\) has been freed by \(t\).
- If there exists a version \(v\) that \(u\) belongs to for which collect\((v)\) has not been called by time \(t\), then \(u\) has not been freed by \(t\).

**The collect Algorithm.** We now present a collect algorithm and show its correctness and efficiency. Path-copying causes subsets of the tuples to be shared among versions. To collect the correct tuples, we use reference counting (RC) [22, 37] for enabling safe garbage collection. Each object maintains a count of references to it, and when it reaches 0, it is safe to collect. Since we use a PLM, the memory graph is acyclic. This means that RC allows collecting everything [37]. In our model, we maintain reference counts for each tuple \(x, x.\text{ref}\), which records the number of “parents” of a node \(x\) in the memory graph. According to a tuple\((v)\) operation creating a tuple \(x\) increments the reference counters of all children of \(x\). We note that tuple\(_v\) can be called only by the writers’ user code when it copies a path. The counts are incremented only by the writers, but can be decreased by any release\(_p\) operation. A newly-created tuple \(u\) has counter 0. Later, when a transaction (reader or writer) executes a collect\(_x\) of a version starting from tuple\(_x\), it first decrements the count of \(x\). Only if the count of \(x\) has reached zero, \(x\) gets freed, and all children of \(x\) are collected recursively. If \(x\)’s counter is more than one, the collect\(_x\) operation terminates since the counts of its descendants will not be decreased then.
Algorithm 4: The Precise, Safe and Wait-free Algorithm for the Version Maintenance Problem

```c
enum VStatus {usable, pending, frozen};
struct Version{
  int timestamp;
  int index; }
struct VersionStatus {
  Version v;
  VStatus status; }
struct Announcement {
  Version v;
  VersionStatus s; }
struct Data* {
  int index;
  VersionStatus s; }

bool set(int k, Data* data) {
  Version oldVer = A[k].v; //the version you acquired
  Version newVer;
  for(int i = 0; i < P; i++) {
    if (A[i] == empty, usable) {
      newVer = V(timestamp+1, i);
      if(CAS(A[i], empty, usable), (newVer, usable)) {
        D[i] = data;
        break;
      }
    }
  }
  if (oldVer != V) {
    if( (oldVer == usable) {
      CAS(A[i], a, (v, false)); } )
    return result;
  }
  S[i] = (empty, usable); }

return D[v.index];
```

Pseudocode for nth() and collect() operation is given in Algorithm 5. We use an array of length l in each tuple x to store the l elements in this tuple (x.ch[]). Inc and dec denote atomic increment and decrement operations. We leave this general on purpose. The simplest way of implementing the counters is via a fetch-and-add object. However, we note that this could introduce unnecessary contention. To mitigate that effect, other options, like dynamic non-zero indicators, can be used.

The result of this section is summarized in Theorem 4.2.

**Theorem 4.2.** Our collect algorithm (Algorithm 5) is correct and takes O(S + 1) time where S is the number of tuples that were freed.

We show the proof of Theorem 4.2 in the full version of this paper [11]. Intuitively, this is because tuples have a constant number of pointers and we only recursively collect any of those pointers if we free the tuple (the count has gone to zero). We can therefore charge the cost of visiting the child against the freed parent.

5 IMPLEMENTING TRANSACTIONS

We now present our transaction system, and show that by plugging in our Version Maintenance algorithm and underlying functional data structures with correct GC, we can get an effective and efficient solution. Read and write transactions are implemented as shown in

```c
var = int or Tuple
struct Tuple { var, ch[]; int ref; }
Tuple tuple(var x) {
  Tuple y = alloc(Tuple);
  y.ref = 0;
  for(int i=0; i<11; i++) {
    y.ch[i] = x[i];
  }
  if(x[i] is Tuple)
    inc(x[i].ref);
  void nth(Tuple x, int i){
    return x.ch[i];
  }
void collect(var x) {
  if(x is int)
    return;
  int c = dec(x.ref);
  if(c>0) {
    var* tmp[11];
    for(int i=0; i<11; i++)
      tmp[i] = nth(x, i);
    free(x);
  }
}
```

Figure 1. We assume all user code works in the functional setting as described in Section 2. The user code takes in a pointer to a version root v, and may access (but not mutate) any memory that is reachable from v. The writer uses path-copying, as standard in functional data structures, to construct a new version. It then can commit the version with the set operation. Here we assume that the write transaction retries if the set fails (i.e., another concurrent write transaction has succeeded). Importantly the user code is unchanged from the (functional) sequential code. A read transaction is active until the last instruction of its user code, and a write transaction is active until the linearization point of its successful set operation. Transactions are live until the last instruction (after the release and GC).
5.1 Correctness and Preciseness

An instantiation of this framework consists of two important parts: (1) a linearizable solution, \( M \), to the version maintenance problem defined in Section 3, and (2) a correct collect function. We show that combining them together yields strict serializability, and safe and precise GC.

Theorem 5.1 (Strictly Serializable). Given a linearizable solution to the version maintenance problem, our transactional framework is strictly serializable.

For proving Theorem 5.1, we define a serialization point for each transaction that is within its execution interval.

Definition 5.2. The serialization point, \( s \), of a transaction \( t \) is:
- If \( t \) is a read transaction, then \( s \) is at the linearization point of \( t \)'s call to \( M.\text{acquire}() \).
- If \( t \) is a write transaction, then \( s \) is at the linearization point of \( t \)'s call to its successful \( M.\text{set}() \).

A proof is given in [11]. Intuitively, we show that if we sequentialize any given history according to these serialization points, it is equivalent to some sequential transactional history.

Theorem 5.3 (Safe and Precise GC). Given a linearizable solution to the version maintenance problem and a correct collect function, our garbage collection is safe and precise.

A full proof is given in [11]. Intuitively, the garbage collection is safe because \( \text{collect}(v) \) is called only when a release returns \( v \), meaning that \( v \) is no longer live. It is precise since if the release is the last one on the transaction's version, the precise Version Maintenance solution will return that version, and any tuples in the version that are not shared with other versions will be collected while the transaction is still live. Therefore no version that is no longer live will survive past the lifetime of the last transaction that releases it.

5.2 Delay and Contention

Here we prove bounds on delay and contention experienced by transactions assuming we use the wait-free algorithm for the version maintenance problem (Section 3.1), and our collect function (Section 4). A summary of the results is shown in Table 1.

Theorem 5.4 (Step Complexity). With our transactional system using the PSWF algorithm for Version Maintenance,

- all read transactions are delay-free,
- all single-writer transactions have \( O(P) \) delay, and
- all write transactions are lock-free.

Furthermore, for single-writers, the time complexity of the garbage collection across a sequence of transactions is bounded by the number of unique tuples used across all versions.

Proof. The proof follows almost directly from previous theorems 3.4 and 4.2. In particular, a read-transaction is active during the acquire and the user code. The acquire takes \( O(1) \) time by Theorem 3.4, and the user code requires no extra time since the code is not changed from the original sequential code. The transaction is therefore delay-free. A write transaction is active during the acquire, user code and until the end of a successful set.

The cost of acquire is \( O(1) \), the cost of set is \( O(P) \) and the user code takes no more time than it would sequentially. If there is no concurrent writer it will succeed on the first try and hence have delay \( O(P) \). If concurrent with other writers it can only fail and restart if some other writing transaction succeeds. Hence it is lock-free.

In the single-writer context, all values are successfully written and hence the number of tuples needed to collect is bounded by the tuples that appear across all versions. By Theorem 4.2 each takes constant time to collect.

Theorem 5.5. For the single-writer setting, all shared-memory operations except inside the garbage collector have \( O(1) \) amortized contention.

Proof. This follows the bounds on contention in Theorem 3.5 for acquire, set, and release. Each has amortized contention proportional to its time complexity. Furthermore in the single-writer context, only a single transaction is allocating and incrementing reference counts at any time. However, in the garbage collection there can be contention when decrementing reference counts.

5.3 Discussion about Functional Data Structures

The important features of the functional code for our purposes is that it is fully persistent and safe for concurrency, both by default. As previously mentioned, persistence can also be achieved by using version lists on each object [13, 39, 46, 51, 56]. This requires modifying every read and write, and can asymptotically increase the time complexity of user code. There has been theoretical work on efficiently supporting version-list based persistence based on node splitting [24]. This approach, however, has several drawbacks in our context. Firstly it requires at most a constant number of pointers to all objects. This would disallow, for example, even having the processes point to a common object. Secondly, it is not safe for concurrency. Making it safe would be an interesting and non-trivial research topic on its own. Thirdly, the approach does not address garbage collection—it assumes all versions are maintained. Again, adding garbage collection would be an interesting research topic on its own. Finally, constant time operations are only supported for what is called partial persistence—i.e. a linear history of changes. Supporting lock-free writers seems to require that multiple writers simultaneously update their versions, which requires what is called full persistence, which allows for branching of the history.

We note that a disadvantage of functional data structures as compared to version lists is that they sequentialize write transactions even when on different parts of a data structure. With version lists, if two transactions are race-free (the set of objects that one writes is disjoint from the set that the other reads and writes), then they can proceed in parallel and serialize in either order. For this reason, we believe our approach is best suited either in situations when the transaction load is dominated by readers, or when the updates can be batched, as described in our experiments. As mentioned in the introduction, due to dependences it is impossible to bound the delay for writers independently of the other concurrent writers. It might be possible, however, to bound delays relative to inherent dependences—i.e., the delay is no more than forced by a dependence.
6 OTHER VM ALGORITHMS

In this section, we present three additional solutions to the Version Maintenance problem. One solution is based on Read-Copy-Update RCU [44] and the other two are based on widely used memory reclamation techniques: Hazard Pointers (HP) [45] and Epoch Based Reclamation (EP) [27]. These solutions are simple to describe, but have various drawbacks. The HP and EP based solutions are not precise. RCU leads to a precise solution, but writers block waiting for readers. Researchers have proposed numerous extensions to the original HP and EP techniques [3, 19, 21, 62]. Some of these directly translate to new ways of solving the VM problem. Our PSWF algorithm can be understood as a wait-free and precise extension of the HP based algorithm. We experimentally compare these version maintenance strategies in Section 7.1.

Read-Copy-Update (RCU). The basic RCU interface provides 3 methods: read_lock, read_unlock, and synchronize. read_lock and read_unlock mark the beginning and end of read-side critical sections. synchronize blocks until all the currently active read-side critical sections have completed. Note that synchronize only needs to wait for the read-side critical sections that existed at the start of its execution.

The RCU-based acquire method calls read_lock and then reads and returns the current version. The set method updates the current version using a CAS (similar to the PSWF algorithm). If the CAS succeeds, it remembers the old version. If release does not follow a successful set, it simply calls read_unlock and returns the empty set. Otherwise, it also has to call synchronize and return the old version to be garbage collected. The downside of RCU is that write transactions have to wait for read transactions and return the current version. A release operation updates the current version using a CAS (similar to the PSWF algorithm). If the CAS succeeds, it returns all the versions retired during that epoch. An acquire operation simply reads and announces the current epoch, and then reads and returns the current version. A release operation reads the current epoch and scans the announcement array. If everyone has announced this epoch, it tries to increment the current epoch with a CAS. If the CAS succeeds, it returns all the versions retired 2 epochs ago. Since everyone has announced the previous epoch, these versions cannot be accessed anymore. In all other cases, the release operation returns an empty list. It is only necessary to maintain a set of retired versions for the last 3 epochs.

To reduce the number of times we scan the announcement array, we only do this for release operations that follow a successful set operation. All other release operations are allowed to return right away. This optimization increases the number of uncollections by at most 1.

7 EXPERIMENTS

In this section, we study the performance of our approach using ordered maps implemented with balanced binary trees. For the ordered maps we use the C++ PAM library [59] since it already supports functional tree structures, and has a reference counting collector. For the experiments, we have implemented five versions of the Version Maintenance: our PSWF algorithm, our algorithm without helping, an imprecise version based on epochs, an imprecise version based on hazard pointers, and a blocking version based on RCU. We do not compare to general purpose software transactional memory systems since previous results show they are not competitive to direct concurrent implementations [29].

We run two types of experiments. The first studies query and update operations under a single-reader multi-reader concurrent setting. The experiments are designed to understand the overheads of the different Version Maintenance algorithm and how much garbage they leave behind. The second type measures the throughput of concurrent operations on functional trees, comparing to five existing trees (or skip lists). It uses batching for our functional tree structure. The goal is to understand the overhead of using functional trees.

Due to space limitation, we present more results and analysis in the full version of this paper [11].

Setup. For all experiments, we use a 72-core Dell R930 with 4 x Intel(R) Xeon(R) E7-8867 v4 (18 cores, 2.4GHz and 45MB L3 cache), and 1Tbyte memory. Each core is 2-way hyperthreaded giving 144 hyperthreads. Our code was compiled with g++ 5.4.1 with the Cilk Plus extensions. We compile with -O3. We use numactl -i all in all experiments, evenly spreading the memory pages across the processors in a round-robin fashion. All the numbers are taken by averaging of 3 runs. In experiments, we use “threads” to refer to “processes” as we use in our theoretical analysis.

7.1 Evaluating the VM Algorithms and GC

In this section, we experiment with five different Version Maintenance algorithms: our precise, safe and wait-free algorithm from Section 3 (PSWF), our algorithm without helping (which only guarantees lock-freedom, referred to as PSLF), a hazard-pointer-based algorithm (HP), an epoch-based algorithm (EP), and an RCU-based algorithm (RCU). The implementation of the latter three is discussed in Section 6. We note that PSWF, PSLF and RCU guarantee precise
garbage collection, while EP and HP do not. RCU guarantees that at any point there are at most two live versions, but will block writers if there are readers working on the old version. HP, EP, and our PSWF algorithm are non-blocking.

We use the functional augmented tree structure in PAM as the underlying data structure. We use integer keys and values, and conduct parallel range-sum queries while updating the tree with insertions. Each query asks for the sum of values in a key range with inserts. The initial tree size is \( n = 10^5 \). We use \( P = 141 \) threads to invoke concurrent transactions, among which one thread continually commits updates, each containing \( n_u \) sequential insertions, and 140 threads conduct queries, each containing \( n_q \) range-sum queries. We control the granularity of update and query transactions by adjusting \( n_u \) and \( n_q \), respectively. We set the total running time to be 15 seconds, and test different combinations of update and query granularity. We keep track of the number of live versions before each update, and report the maximum number of versions. The results are shown in Table 2 and Figure 6.

**The number of live versions.** The number of live versions for all five algorithms in different settings is shown in Table 2. Figure 6 shows the maximum live versions of the five VM algorithms, with different update granularity when \( n_q = 10 \). The general trends for all five algorithms are similar. When \( n_u \) is large or \( n_q \) is small, there are few versions live. This is because when updates are less frequent or queries finish fast, most queries will catch recent versions. When \( n_u \) is small or \( n_q \) is large, the number of live versions gets larger. This is because when new versions are generated frequently, or queries take a long time, it is more likely for queries to be behind the current version, and keep more old versions live.

We now compare the five VM algorithms. The maximum number of live versions for HP is always \( 2P = 282 \). For EP, when \( n_u \) is large, the number of live versions is reasonable and mostly below 100. However, for frequent updates, the number of versions can reach up to 1000 (see Figure 6), because queries cannot catch up with the latest version. Many recent (but not current) versions cannot be collected, even if no queries are working on them. Theoretically the epoch-based algorithm can leave an unbounded number of versions behind. RCU keeps only 1 version before set since the writer will wait to collect the old version before generating a new version. Although the amount of garbage is small, the writer is blocked and update granularity is low as we will show later in this section. For our PSWF algorithm, the number of total versions is at most 141 for small \( n_u \) and large \( n_q \). This case is possible but rare to occur. In the settings we shown in this paper, the maximum number of versions is within 100. In most of the cases, the maximum of living versions is around 10, which is 1/14 of the total query threads. Because our GC is precise, all out-of-date versions are collected immediately. The helping scheme is our PSWF does not affect much of the number of maximum versions. For all tested setting, the number of versions kept by our PSWF algorithm is only 1.5-83\% less than EP, and about 7-120\% less than HP.

The throughput of queries and updates. We report the query and update throughput (millions of queries/updates per second) for different settings in Table 2. We compare the throughput numbers for base cases when no VM (and thus no GC) algorithms are adopted, noted as “Base” in the Tables.

Generally, from Table 2 we can see that introducing a VM algorithm always lowers the throughput of queries and updates. This is not only because of the overhead in maintaining versions, but also from the possible GC cost. For both updates and queries, we do not see a significant difference between our PSWF algorithm and PSLF algorithm. Generally this means that in practice, it is very rare that the writer needs to help the readers a lot. We do see a more notable difference in extreme cases (e.g., \( n_u = 1 \) [11]).

**Queries.** For all the five algorithms and all the four settings, the overhead of introducing GC and VM algorithms is around 10\% for queries. The five VM algorithms have comparable performance. RCU usually has much better query performance, this is possibly because all the queries of RCU are working on the same version, and thus leading to better locality.

**Updates.** Generally, larger \( n_u \) results in better update throughput. There are mainly two reasons. Firstly, batching more updates in one transaction reduces the overhead in calling acquire, set and release for version maintenance. Secondly, larger update transactions allow more query threads to catch more recent versions, and thus a larger fraction of the current version will appear in cache, making updates faster. The overhead of introducing GC and VM algorithms is within 20\% for our PSWF algorithm, but can be more for the other algorithms. Our algorithms are always the best among all the algorithms in terms of update throughput. It is likely because
for HP, EP and RCU, the writer is responsible to do all GC work, while in PSWF, queries and updates share the responsibility of GC. Note that although RCU has the best performance in queries, it has much lower update performance than the others, because the writer can be blocked by unfinished queries.

Overall. Generally, our PSWF algorithm is comparable to the EP and HP, and slightly slower than RCU in queries, but is always much faster in updates than all the other implementations. As mentioned, this is mostly due to the difference in GC responsibility. Therefore, our algorithms have the best overall performance.

7.2 Functional Concurrent Operations

In this section test the throughput of concurrent operations on the functional tree in PAM.

Concurrent Operations with Batching. We compare the functional tree to several state-of-the-art concurrent data structures: skiplists [54], OpenBW trees [60], Masstree [40], B+trees [60] and concurrent Chromatic trees [17, 18] (all in C++). For all structures we turn GC off since we are interested in the performance of the trees and not the GC. We use the Yahoo! Cloud Serving Benchmark (YCSB) microbenchmarks, which have skewed access patterns (Zipfian distributions) to mimic real-world access patterns. We test YCSB workloads A (read/update, 50/50), B (read/update, 95/5) and C (all read). The original dataset (before updates) has 5 × 10^7 elements, and each workload contains 10^7 transactions. We use 64-bit integers.

For PAM we use batching to collect concurrent updates so they can be updated in parallel using single-writer. The batching works by accumulating update requests in a buffer and when there are a sufficiently many, applying them using PAM’s multi-insert function, which is a parallel divide-and-conquer algorithm [15]. The batch size is controlled so the latency for an update is no more than 50ms. More details on batching are given in the full version of this paper. The reads (finds in the tree) do not need to be batched since any number of readers can run concurrently.

The results on operation throughput are presented in Figure 7. In all the three workloads, our implementation outperforms the best of the others by 20%-300%. There are a few factors contribute to the good performance of our implementation. Firstly, the code for a query is just a standard tree search with no additional cost for synchronization. Secondly, since the code for the batched updates uses a parallel divide-and-conquer algorithm for each batch, it generates no contention between writes.

We note that the comparison is not apples-to-apples. Due to batching, our updates have higher latency than the others. This will not be appropriate in some applications. On the other hand, our approach allows multiple operations to be applied atomically, while the others only support atomically at the granularity of individual operations.

Inverted Index Searching. We test the functional tree on searching an inverted index [55, 63] to show the overhead of read/write transactions on functional data structures. We maintain a tree for the inverted index. Read-only searching queries come in concurrently, and a single thread is responsible for adding new documents to the index. The update can be done in parallel since multiple words are added simultaneously. Due to page limitation, we omit the details here. More results are shown in the full version of this paper [11].

8 RELATED WORK

Multiversioning has been studied extensively since the 70s [13, 51, 56]. However, most previous protocols, like multiversion timestamp ordering (MVTO) [56] and read-only multiversion (ROMV) [49, 61] are time-stamp based, maintaining version lists for every object, which are traversed to find the object with the proper timestamp. This approach inherently delays user code since version lists can be long. It also complicates garbage collection. Kumar et al. [39] revisit the MVTO protocol and develop a concrete algorithm with GC that has similar properties to ours if the GC is applied frequently enough. However this requires scanning whole version lists for objects and requires locks. Also in their algorithm the writer can still delay readers and the readers can abort the writer. As far as we know no work based on multiversioning with version lists has shown bounds on time or space.

Pérelman, Fan and Keidar [52] showed resource bounds for multiversion protocols. They define the notion of MV-permissiveness, which means that only write transactions abort (or restart), and only if they conflict. They also define useless prefix (UP) GC, which is similar but slightly weaker than our notion of precise GC (it only collects proper prefixes of the versions). They describe an algorithm that is MV-permissive and satisfies UP GC. They do not give any time bounds—the delay could take time that is a function of data structure size and number of processes, even when there is a single writer, since the approach is based on copying an old value to all previous active versions.

Beyond RCU [44], the read-log-update (RLU) protocol also supports two versions such that readers can read an old version, while the writer updates the current version [41]. The RLU allows readers to see the currently updated version, but still blocks before the next version can be updated until all processes reach a quiescent period. Attiya and Hillel [6] suggest a similar idea that allows readers to proceed while blocking writers (even a single writer).

Path-copying is a default implementation in functional languages, where data cannot be overwritten [48]. Similar techniques have been used for maintaining multiversion B-tree or B+tree structures or their variants [7, 58], and is used in real-world database systems like LMDB [1], CouchDB [4], Hyder [14] and InnoDB [28], as well as many file-systems [16, 20, 23, 36, 57].

Some techniques in our algorithm can also be found in wait-free universal construction algorithms [25, 31, 33]. More details can be found in the full version of this paper.

9 ACKNOWLEDGEMENT

This work was supported in part by NSF grants CCF-1408940, CCF-1533858, and CCF-1629444.

REFERENCES
