Abstract

This paper presents a generic approach for deriving detectably recoverable implementations of many widely-used concurrent data structures. Such implementations are appealing for emerging systems featuring byte-addressable non-volatile main memory (NVMM), whose persistence allows to efficiently resurrect failed threads after crashes. Detectable recovery ensures that after a crash, every executed operation is able to recover and return a correct response, and that the state of the data structure is not corrupted.

Our approach, called Tracking, amends descriptor objects used in existing lock-free helping schemes with additional fields that track an operation’s progress towards completion and persists these fields in order to ensure detectable recovery. Tracking avoids full-fledged logging and tracks the progress of concurrent operations in a per-thread manner, thus reducing the cost of ensuring detectable recovery.

We have applied Tracking to derive detectably recoverable implementations of a linked list, a binary search tree, and an exchanger. Our experimental analysis introduces a new way of analyzing the cost of persistence instructions, not by simply counting them but by separating them into categories based on the impact they have on the performance. The analysis reveals that understanding the actual persistence cost of an algorithm in machines with real NVMM, is more complicated than previously thought, and requires a thorough evaluation, since the impact of different persistence instructions on performance may greatly vary. We consider this analysis to be one of the major contributions of the paper.
1 Introduction

The availability of byte-addressable non-volatile main memory (NVMM) has increased the interest in the crash-recovery model, in which failed threads may be resurrected after the system crashes. Of particular interest is the design of recoverable concurrent objects (also called persistent [10, 12] or durable [46]), whose operations can recover from crash-failures. It is also important to be able to tell after recovery whether an operation was completed and if so, what its response was, a property called detectable recovery [3, 25]. This is because, in some applications, re-executing completed operations may result in undesirable outcomes. Moreover, detectable recovery allows a programmer to easily predict which operations’ responses are correct in systems with crashes, enhancing ease of programming in such systems.

As in database systems, detectable recovery can be supported by precisely logging the progress of computations to non-volatile storage, and replaying the log during recovery. However, logging imposes significant overheads in time and space. In the context of concurrent data structures, full-fledged logging is not needed and the progress of an operation can be tracked individually. Moreover, many lock-free implementations already encompass such tracking mechanisms, which can be easily adapted to support detectable recovery. This leads to the Tracking approach for designing recoverable objects, based on explicitly maintaining information tracking an operation’s progress as it executes, stored in non-volatile memory. This information allows a thread to decide, upon recovery, whether the operation’s effect has become visible to other threads, and if it did, determine the response of the operation. (See Section 3.)

In many cases, Tracking requires small changes to the original code. It significantly saves on the cost (in both time and memory) by simply tracking specific stages of the execution of an operation. Even more, this can often be piggybacked on information already tracked by lock-free concurrent data structures, within operation descriptors. This means that operations efficiently maintain and persist sufficient information for recovery, and that the corresponding recovery code infers whether the operation took effect before the failure, in which case its response value is computed and returned. These properties are what make our approach attractive.

Tracking is widely applicable—it can be used to derive recoverable versions of a large collection of concurrent data structures. We have applied it to derive a linked list (based on [29], in Section 4), a binary search tree (based on [19]), and an exchanger object (based on [43]). Tracking informs how persistence instructions [32] (such as pwb, pfence and psync, which are implemented with flushes and fences) should be inserted for ensuring an implementation’s correctness in an efficient manner, when cache memories are volatile and their content is lost upon a failure [32] (see Section 3).

We provide an experimental analysis (Section 5) which compares the performance of Tracking with all other generic schemes we are aware of that can be used for deriving detectable recoverable data structures, as well as with many previous, publicly-available, schemes that ensure only durable linearizability [15, 16, 41]. For our experimental analysis, we took on the challenge of implementing (in C++) a hand-tuned, highly-optimized version of detectably recoverable linked lists using Capsules [6]. Capsules is a general transformation for achieving detectability, which can be applied to a concurrent algorithm that uses only read and CAS operations in order to make it detectably recoverable. The optimized version (CAPSULES-OPT) we implemented exhibits the best performance among the competitors. Experiments show that in most cases, Tracking performs better than CAPSULES-OPT and all other competitors. However, closer inspection revealed that Tracking executes more persistence instructions than CAPSULES-OPT. This came as a surprise and caused us to perform a more comprehensive experimental analysis to understand the reasons. The analysis revealed the following interesting points: 1) The impact of persistence fences (psync and pfence) in the machine with Intel Optane we are working on is negligible; after removing all such instructions from our algorithms, we do not see any important impact in performance. 2) Different flush instructions have differ-
ent impact in performance. Specifically, those flushes that are applied on a private non-volatile variable of a thread (such variables are usually used by the thread for tracking its progress and flushing them is necessary for ensuring detectability), or on newly allocated data that is not yet shared, are cheap, whereas others that access shared highly-contended variables are expensive.

Our experimental analysis categorizes code lines of persistence instructions based on the impact they have in performance at different interleavings. By doing so, we see that the persistence instructions in Tracking are mostly cheap, whereas Capsules performs mostly expensive persistence instructions. This results in a higher persistence cost for Capsules than for Tracking. Our analysis reveals that to fully understand the performance of a persistent algorithm, experiments providing quantitative data (i.e., measuring the number of flushes and fences) are not enough. More work is necessary to figure out from which specific flush instructions the actual persistence cost comes and try to avoid them whenever possible. We consider the above insights and the experimental scheme we propose to be major contributions of the paper.

Summarizing, the main contributions of this paper are:

- We propose Tracking, a new transformation for deriving detectably recoverable implementations of concurrent data structures.
- We show how persistence instructions can be added in systems with volatile caches in a manner that enhances efficiency and scalability.
- We apply Tracking to get new detectably recoverable implementations of several data structures.
- We provide an experimental analysis to compare with all existing detectably recoverable transformations we are aware of. Tracking exhibits better performance than all its competitors in most cases.
- The experimental analysis reveals that the persistence instructions should be categorized based on the impact they have in performance. We provide a novel experimental scheme to do so and explore the characteristics of persistence instructions in different categories.

2 System Model

We consider a system of asynchronous crash-prone threads which communicate through base objects supporting atomic read, write, and Compare&Swap (CAS) primitive operations.

We assume that the main memory is non-volatile, whereas the data in the cache or registers are volatile. Thus, writes can be persisted to the non-volatile memory using explicit flush instructions, or when a cache line is evicted. Under explicit epoch persistency [32], a write-back to persistent storage is triggered by a persistent write-back (pub) instruction; a pub flushes all fields fitting in a cache line. The order of pub s is not necessarily preserved. A pfence instruction orders preceding pubs before all subsequent pubs. A psync instruction waits until all previous pubs complete the write backs. For each location, persistent write-backs preserve program order. We assume the Total Store Order (TSO) model, supported by the x86 and SPARC architectures, where writes become visible in program order.

At any point during the execution of an operation, a system-wide crash-failure (or simply a crash) resets all volatile variables to their initial values. Failed threads are recovered by the system asynchronously, independently of each other; the system may recover only a subset of these threads before another crash occurs.

A thread q invokes Op to start its execution; Op completes by returning a response value, which is stored to a local variable of q (and thus it is lost if a crash occurs before q persists it). A recoverable operation Op has an associated recovery function, denoted Op.Recover, which the system calls when recovering q after a system-failure that occurred while it was executing
Op. The recovery code is responsible for finishing Op’s execution and returning its response. Thread q may incur multiple crashes while executing Op and/or Op.Recover. We assume that the system invokes Op.Recover with the same arguments as those with which Op was invoked when the crash occurred. For each thread q, we also use a non-volatile private variable CP_q, that recoverable operations and recovery functions use for managing check-points in their execution flow. When q invokes a recoverable operation Op, the system sets CP_q to 0 just before Op’s execution starts. CP_q can be read and written by recoverable operations (and their recovery functions). CP_q is used by q in order to persistently report that the execution reached a certain point. The recovery function can use this information in order to correctly recover and to avoid re-execution of critical instructions such as CAS.

Op is completed either directly or when, after one or more crashes, the execution of the last instance of Op.Recover invoked for q is complete. An execution is durably linearizable [33], if the effects of all operations that have completed before a crash are reflected in the object’s state upon recovery. In addition to durable linearizability, we ensure detectability [25]: it is possible to determine, upon recovery, whether the operation took effect, and if it did, its response value. Detectable recoverability ensures durable linearizability and detectability.

A recoverable implementation is lock-free, if in every infinite execution produced by the implementation, which contains a finite number of system crashes, an infinite number of operations complete.

3 The Tracking Approach

Consider an implementation of a data structure that is represented as a set of nodes, each with data fields and pointers to other nodes in the data structure. Figure 1(a) provides an example of a sorted linked list with two sentinel nodes implementing the set {3, 8, 15, 27}.

We first provide an overview of how Tracking works. Each operation Op is associated with an operation descriptor, tracking the information needed to complete Op. Moreover, each node is augmented with a special info field, containing a pointer to an operation descriptor, which may be tagged. Intuitively, when an operation Op tags a node, it is like if it puts a “soft” lock on it to ensure that it will not be updated by other operations until Op completes. The node may be later untagged, thus releasing the soft lock on it.

Briefly, in Tracking, the execution of an operation Op, initiated by a thread q, takes place in phases. The first phase is the gather phase, where q searches in the data structure to find the nodes that may be affected by Op, i.e., those nodes that Op will attempt to update or delete, nodes that need to be tagged for performing these updates and deletions, as well as nodes that contain values the operation will return or are needed to determine the operation’s response. This set of nodes is called the AffectSet of Op. In our example linked list, a successful insert (or delete) affects the last two nodes it accesses during its search, and a Find (or an unsuccessful update) affects only the last node it accesses.

Thread q then performs Op’s helping phase, where it checks if any node nd in Op’s AffectSet is already tagged by another operation Op’. If this is so, q uses the information in Op’’s operation descriptor to help Op’ complete. Next, q proceeds to the tagging phase of Op, where it attempts to tag each of the nodes in AffectSet, storing a pointer to Op’s descriptor in the info fields of these nodes (using CAS). After a successful tagging phase, Op can proceed to its update phase, where all its updates are applied. Next, Op’s response is recorded in Op’s descriptor, and finally, the nodes in Op’s AffectSet are untagged (cleanup phase). The tagging phase may not succeed if some node nd in Op’s AffectSet is already tagged by another operation Op’. Then, q untags all nodes it has already tagged and re-starts the execution of Op.

To support detectable recovery, when q recovers from a crash that occurred while executing Op, its recovery code must be able to access Op’s descriptor. This is achieved by allocating, for

1 System support is necessary for designing detectable algorithms [5].
Algorithm 1: Tracking - Op and Op-Recover (code for thread $q$)

Procedure Op (args)

1. Info *opInfo := new Info ()
2. RD$_q$ := Null
3. pbarrier (RD$_q$)
4. CP$_q$ := 1
5. pwb (CP$_q$); psync
6. while true do
   7. Gather Phase
      8. Traverse the data structure gathering pairs ($nd$, $ndinfo$) of pointers to those nodes (and to their info fields) that Op will affect, e.g. will try to update or delete, as well as nodes that contain the values the operation will return (the info field of each node is gathered on the first access to it)
      9. AffectSet := list of all such pairs
   10. Helping Phase
      11. if there is a tagged ndinfo field in a pair of AffectSet
      12. Help(ndinfo)
      13. continue
      14. WriteSet := list of structs of type Update containing those fields of nodes from AffectSet that have to change, together with an old and a new value for each of them to perform the change using CAS
      15. NewSet := list of newly allocated nodes that are necessary to execute the operation, tagged with opInfo
      16. *opInfo := (Op, AffectSet, WriteSet, NewSet, ⊥) // ⊥ for result
      17. if WriteSet = ∅ and AffectSet contains only one element
      18. opInfo → result := response determined based on opInfo
      19. pbarrier (*opInfo, NewSet)
      20. pwb (RD$_q$); psync
      21. if WriteSet = ∅ and AffectSet contains only one element
      22. return opInfo→result
      23. else re-invoke Op
   24. if opInfo→result ≠ ⊥ then return opInfo→result
   25. opInfo := new Info ()

Procedure Op-Recover (args)

27. Info *opInfo := RD$_q$
28. if CP$_q$ = 0 or opInfo = Null then Re-invoke Op (args)
29. HELP(opInfo)
30. if opInfo→result ≠ ⊥ then return opInfo→result
31. else re-invoke Op

every thread $q$, a designated persistent recovery data variable, RD$_q$, that stores a reference to the descriptor of its last operation. To ensure detectable recovery, thread $q$, and every thread helping Op, sets the result in the descriptor before untagging the relevant nodes. Upon recovery, $q$ reads a reference to its last descriptor from RD$_q$ and uses it to complete its last operation Op. If the result field of the descriptor is set, Op took effect, and $q$ returns its value. Otherwise, Op did not take effect and it can be restarted. Even if Op performed changes that have been later obliterated by other operations, the result field of Op would still be set.

Many lock-free implementations of data structures (e.g., [4, 18, 19, 23, 48]) follow a similar tracking approach to enable helping and ensure global progress. They associate an operation descriptor with each update operation, tracking the progress of the update by storing sufficient information to allow its completion by concurrent operations. This scheme goes a long way towards making a data structure recoverable: updates are idempotent and not susceptible to the ABA problem, since they must ensure that an update is done exactly once, even if several
Algorithm 2: Tracking - HELP (code for thread q)

Procedure HELP (Info *opInfo)

Tagging Phase
⟨nd, ndinfo⟩ := head of opInfo→AffectSet
while nd ≠ NULL do
  res := CAS(nd→info, ndinfo, getTagged(opInfo))
  pub (nd→info)
  if res ≠ ndinfo and res ≠ getTagged(opInfo) then
    ⟨nd, ndinfo⟩ := previous element in AffectSet
  end
  while nd ≠ NULL do
    CAS(nd→info, getTagged(opInfo), getUntagged(opInfo))
    pub (nd→info)
    ⟨nd, ndinfo⟩ := previous element in AffectSet
  end
  return
⟨nd, ndinfo⟩ := next element in AffectSet
psync

Backtrack Phase
⟨nd, ndinfo⟩ := previous element in AffectSet
while nd ≠ NULL do
  CAS(nd→info, getTagged(opInfo), getUntagged(opInfo))
  ⟨nd, ndinfo⟩ := previous element in AffectSet
  return
psync

Update Phase
foreach Update structure st in WriteSet do
  perform the update by executing CAS based on information contained in st
  pub (updated field)
  opInfo→result := response of the operation described by opInfo
  pub (opInfo→result); psync

Cleanup Phase
foreach node nd in (AffectSet ∪ NewSet) which is still part of the data structure do
  CAS(nd→info, getTagged(opInfo), getUntagged(opInfo))
  pub (nd→info)
  psync

threads attempt to concurrently help it complete. Tracking takes advantage of such helping mechanisms to provide detectable recovery: it piggybacks the data needed for persistence within the descriptors that are already used by the helping mechanism. This saves in cost and results in small changes to the original code.

3.1 Detailed description

High-level pseudocode for Tracking appears in Algorithms 1 and 2; code in blue, dealing with recoverability, and code in red, dealing with an optimization for read-only operations, are explained below. We implement tagging by setting the least significant bit of info. The node pointer to the descriptor is tagged when it is first installed in nd. A node is tagged if its info field is tagged. GetTagged (getUntagged) returns a tagged (untagged) version of its argument without changing its value.

An execution of an operation Op by a thread q goes through one or more attempts, each of which is an iteration of a while loop, until one of them is successful and Op returns. In each attempt, Op first executes its gather phase, computing its AffectSet, which is a set of pairs, each comprised of a pointer to a node and the value of its info field. The AffectSet contains those fields of nodes from AffectSet that need to change, together with an old and a new value for each of them (needed to perform the change using CAS). The NewSet contains all newly allocated nodes by Op that are necessary for applying
its updates (all these nodes are initially tagged with a pointer to \( Op \)'s descriptor). Then, the type of \( Op \), its AffectSet, its WriteSet, its NewSet, and the value \(-\) (which is the initial value for the result field) are stored in \( Op \)'s descriptor. Next, HELP is called with parameter \( opInfo \), a pointer to \( Op \)'s descriptor, to complete \( Op \) itself. When HELP returns, if the result field of \( Op \)'s descriptor is not equal to \(-\), then \( Op \) has been performed and its result is returned; otherwise, a new attempt is started.

HELP applies CAS to try to install \( opInfo \) in every node of AffectSet, in order. If any of these CAS fails or the associated info field is tagged by another operation, then a backtrack phase untags the nodes in AffectSet, in reverse order. After backtracking, HELP returns. If every node of AffectSet is successfully tagged with \( opInfo \), then all changes to the WriteSet are being performed and the result field is updated. Finally, the cleanup phase untags every node of AffectSet and NewSet still in the data structure.

If HELP completes the tagging phase, it returns only after \( Op \) takes effect: all CAS operations are applied to its WriteSet, its result is updated and the cleanup phase is done. Otherwise, \( Op \) does not take effect.

Figure 1(b) shows how the example linked list looks like after the tagging phase of a Delete(15) operation completes successfully. The operation descriptor of Delete(15) contains its AffectSet and its WriteSet (the NewSet is empty since a delete does not allocate any new node, so it is not shown in the figure). The AffectSet contains two pairs, containing information for nodes 8 and 15, respectively. The WriteSet contains a triple, containing the next field of node 8, its old value which is a pointer to node 15, and its new value which is a pointer to node 27. The result field of the descriptor has not yet been set. Figure 1(c) shows the state of the linked list after the cleanup phase of Delete(15) is over. The next field of node 8 has been updated to point to node 27, so node 15 has been deleted. The info fields of nodes 8 and 15 still point to the operation descriptor of Delete(15) but node 8 is now untagged, whereas node 15 will remain tagged forever. The result field of the descriptor now contains the value true.

We make the following assumptions on the original implementation: (a) It does not store the same value into a shared variable more than once, and therefore it handles the ABA problem. (b) The nodes in the AffectSet or WriteSet are accessed in the same order. This order can be imposed in many ways and is not necessarily fixed at the beginning of the execution. For
example, in a binary search tree, ordering can be determined by inorder or other traversal orders. (c) \textbf{Help} is idempotent, i.e., its changes are applied exactly once independently of how many times they are performed. This assumption is reasonably general, as many existing concurrent data structures satisfy it (e.g., \cite{[13],[22],[8],[33]}).

\textbf{Help} succeeds in updating the WriteSet, unless some other thread has started the cleanup phase for \textit{Op}, implying that the WriteSet has already been updated. Thus, \textit{Op} succeeds after \textit{q} collects a consistent set of nodes, which do not change until updates on the WriteSet are completed. We linearize \textit{Op} at the beginning of the update phase. At this point, \textit{Op} is guaranteed to complete, and other operations accessing any node in the AffectSet must first help \textit{Op} to complete.

To support detectable recovery, a pointer to the descriptor used in each attempt is stored in \textit{RD}_\textit{q} before \textbf{Help} is called. If the check-point is not set or \textit{RD}_\textit{q} is still \textsc{Null}, \textit{Op} has made no changes and can simply be restarted. Otherwise, the descriptor pointed to by \textit{RD}_\textit{q}, \textit{opInfo}, indicates whether the last attempt of \textit{Op} was successful, and if not, whether it crashed while making changes. Since \textbf{Help} is idempotent, recovery can call \textbf{Help}(\textit{opInfo}) to complete \textit{Op}, in case it is still in progress. When \textbf{Help} returns, either \textit{Op} took effect and \textit{result} stores its response, or it did not and \textit{result} is \perp; in the latter case, \textit{Op} can be re-invoked. It is necessary to call \textbf{Help} first, to deal with the case in which \textit{result} has been written but the operation still needs to clean up, in order to keep the data structure in a consistent state, without nodes tagged by \textit{Op}.

\subsection*{3.2 Lock-freedom}

\textbf{Proof outline.} Since \textbf{Help} is wait-free, as it contains no unbounded loops, recovery is also wait-free.

If in an execution, all processes repeatedly perform failed attempts, then the data structure remains static, and only \textit{info} fields are changed. Note that in this case, after some point, it must hold that for every active operation \textit{Op}, \textit{Op}’s tagging phase does not complete. By inspecting the pseudocode, we see that, otherwise, \textit{Op} would also complete successfully (contradicting our hypothesis that all processes repeatedly perform failed attempts).

Consider a process \textit{q} making infinitely many failed attempts while executing \textit{Op}. Each attempt has a gather phase, followed by a call to \textbf{Help} that fails tagging and performs a backtrack phase. Since only \textit{info} fields are changed, \textit{q} completes the gather phase with the same AffectSet. An attempt fails because some node in AffectSet is tagged by another operation \textit{Op’}. If the process executing \textit{Op’} does not take steps (i.e. it is inactive), then the node is untagged, by \textit{q} or some other process, during a backtrack phase in \textbf{Help}. Thus, only a constant number of attempts fail because a node in AffectSet is tagged by an inactive operation. Other attempts fail due to nodes in AffectSet tagged by operations that keep taking steps (i.e. they are active).

Eventually, we reach a scenario where all operations perform failed attempts, each with the same AffectSet, in which one of the nodes is tagged by another active operation. We now show that the total order assumption prevents such a scenario (of having a livelock). Let \textit{q’} be the process that tagged the latest node in the total order by which processes tag the nodes in their AffectSet. Clearly, no later node in AffectSet of \textit{q’} is tagged by an active operation, and thus \textit{q’} will complete its tagging phase, and its attempt successfully.

\subsection*{3.3 Linearizability}

\textbf{Proof outline.} Intuitively, tagging puts “soft” locks on the nodes in the AffectSet of an operation \textit{Op} (initiated by a process \textit{q}): it ensures that these nodes are not updated (by operations not helping \textit{Op}) until \textit{Op}’s backtrack or cleanup phase completes. The cleanup phase starts only after some process successfully updates \textit{Op}’s WriteSet and the response in \textit{result}. Since \textbf{Help} is idempotent, a node is changed exactly once. It can be shown that the
info field of a node does not hold the same pointer twice whenever it is tagged. Other fields are not prone to ABA, by assumption.

These features ensure that HELP succeeds in updating the write set, unless some other HELP invocation for Op has started the cleanup phase, implying that the WriteSet has already been updated. Thus, Op succeeds after its invoker q collects a consistent set of nodes, which do not change until some process completes the updates on the WriteSet.

We linearize Op at the beginning of the update phase. The above argument implies linearizability, as at this point the operation is guaranteed to complete, and other operations accessing any node in the AffectSet must first help Op to complete. Moreover, detectability follows as well, since a cleanup phase is performed only after Op completes all its updates, and result is updated with Op’s response.

After adding persistence instructions, the correctness argument remains the same, although the following scenario may occur: a tagging phase (for opInfo) may tag all the AffectSet, but some of these tags (not necessarily in order) were not persisted when a crash occurs. At recovery, a different operation may already tagged some of these untagged nodes, and re-tagging may fail. Then, backtracking will only untag a prefix of AffectSet, while other nodes are still tagged (for opInfo). A similar scenario occurs if a process fails during cleanup. These scenarios do not violate durable linearizability or lock-freedom: Since HELP is idempotent, any process that later observes one of these tagged nodes, will fail during tagging and untag the node.

3.4 Persistence instructions

After setting the check-point, allocating a descriptor and storing a reference to it in RDq, a pwb followed by a psync ensures that the data is accessible upon recovery. A pbarrier after initializing RDq ensures that pws are executed in program order. We also insert pwb after every CAS and write in HELP. A psync at the end of every phase persists its changes before the next phase.

An attempt to execute the changes of an operation Op, with a descriptor opInfo, happens after tagging is complete and persisted in HELP(opInfo). A crash before tagging ends may result in an old copy for the info field of some nodes, but in this case, no thread started the update phase using the lost opInfo. Thus, upon recovery, the initiator of Op will call HELP(opInfo) to tag again. Nodes are updated only after all nodes in AffectSet and NewSet are tagged and persisted. Every operation affected by these nodes first completes HELP(opInfo). A node is untagged in the cleanup phase, after all changes of Op and its result field are persisted.

A crash during cleanup may cause an untagged node nd to be tagged, although another operation Op’ might have tagged nd in the meantime. Then, the tagging phase by Op’ is yet to be completed, and both Op and Op’ invoke HELP at recovery time. Therefore, Op must first untag nd before Op’ can tag it again.

3.5 Optimizing for read-only operations

Read-only operations are supported by many concurrent data structures, e.g., FIND in those implementing dictionaries, where the AffectSet contains just a single element; moreover, they determine their response values based on node fields that are immutable. Under these conditions, Tracking can be optimized so that a thread q executing such a read-only operation Op is performed without executing HELP for Op, i.e., by skipping the last three phases of Tracking. Supporting this optimization requires small additions in Algorithm 1 (code in red). However, they affect the way we assign linearization points to read-only operations: A read-only operation Op (that satisfies the optimization condition) is linearized at the point that the info field of the single node added in the AffectSet is read in Op’s last attempt.

To argue about correctness, we provide a simulation proof, where for each execution α of Algorithms 1 and 2 we present a valid execution α’ of Algorithms 1 and 2 which contains the
type Node {
    Key ∪ {−∞, +∞} key
    Node *next
    Info *info
}

type Info {
    {DELETE, INSERT, FIND} opType
    Set AffectSet;
    Set WriteSet;
    Set NewSet;
    boolean result
}

> Initialization:
Shared Node: *head with key −∞, *tail with key +∞.  
head → next points to tail, tail → next points to NULL.
Both info fields are NULL.

Figure 2: Recoverable Linked List types and initialization.

same operations as α and each operation has the same response in both α and α’. Consider a read-only operation Op (that satisfies the optimization condition) executed by some process q in α. We construct α’ by letting q execute solo (i.e., without any other process taking steps concurrently with it), starting from the instruction at which it is linearized, the part of Op (Algorithms 1 and 2) that it is still to be executed in order to complete. Since Op does not change the data structure, the resulting execution (in which Op tags and untags the single node in its AffectSet before any other process takes steps) is a valid execution of Algorithms 1 and 2. To get α’, we apply this technique to all read-only operations that can be optimized.

4 Detectably Recoverable Linked List

We illustrate how to apply Tracking (Algorithms 1 and 2) to get a detectably recoverable linked list. As in our example, the list is sorted in increasing order of keys, with two sentinel nodes, head and tail, holding keys −∞ and +∞. A node nd may be tagged either for update (indicating its next field is about to change), in which case it is untagged after the update completes, or for deletion (indicating it is to be deleted), in which case it remains tagged forever. When nd is tagged, its descriptor contains information necessary to complete the operation that tagged nd. A field opType in the descriptor indicates the operation type (INSERT or DELETE). The data types, shared variables, and initialization values of the algorithm appear in Figure 2.

An instance Op of INSERT(k) (Algorithm 3), executed by a thread q, calls SEARCH during its gather phase (Lines 9–10) to get pointers pred and curr to the nodes between which k should be added, and their info fields. If Op is successful, these are the nodes contained in Op’s AffectSet. Thus, the helping phase (Lines 14–18) simply checks whether these two nodes are tagged and calls HELP if needed.

If the key k to be inserted is already in the list, Op is read-only and behaves like a FIND. In this case, the AffectSet includes just the last node accessed by its search, and we apply the optimization for read-only operations (Lines 21–23 and 31). Otherwise, Op calls HELP of Algorithm 2 (Line 32), after recording the appropriate AffectSet, WriteSet, and NewSet in Op’s Info (Lines 12–13 and 25–27).

DELETE is simpler than INSERT since it does not allocate new nodes. Algorithm 4 provides the pseudocode for DELETE and FIND. FIND(k) is read-only and computes its response based on immutable fields. Moreover, its AffectSet contains just the node pointed by curr. Therefore, FIND is optimized to avoid installing a descriptor. Recovery is achieved in exactly the same way as in Algorithm 1.
Algorithm 3: Recoverable Linked List - INSERT and auxiliary function SEARCH (code for thread \( q \))

**Procedure** boolean INSERT (T key)

1. Node *newcurr := new Node (⊥, NULL, NULL)
2. Node *newnd := new Node (key, newcurr, NULL)
3. Info *opInfo := new Info ()
4. \( RD_q := \perp \)
5. pbarrier (\( RD_q \))
6. \( CP_q := 1 \) \hspace{1cm} // check-point; \( RD_q \) is initialized
7. pwb (\( CP_q \)); psync
8. while true do
9. Gather Phase // search for location to insert
10. \( \langle pred, curr, predInfo, currInfo \rangle := \text{Search(key)} \)
11. if \( curr \rightarrow key = key \) then
12. AffectSet := \{ \( \langle curr, currInfo \rangle \) \}
13. else AffectSet := \{ \( \langle pred, predInfo \rangle, \langle curr, currInfo \rangle \) \}
14. Helping Phase // help other if necessary
15. if isTagged(predInfo) then
16. Help (predInfo); continue
17. else if isTagged(currInfo) then
18. Help (currInfo); continue
19. newcurr := \( \langle curr \rightarrow key, curr \rightarrow next, \text{getTagged(opInfo)} \rangle \)
20. newnd \( \rightarrow \) info := getTagged(opInfo)
21. if \( curr \rightarrow key = key \) then // key in list
22. WriteSet := NewSet := \{ \};
23. opInfo \( \rightarrow \) result := false
24. else
25. WriteSet := \{ \( \langle pred \rightarrow next, curr, newnd \rangle \) \}
26. NewSet := \{ newnd, newcurr \}
27. *opInfo := \{ INSERT, AffectSet, WriteSet, NewSet, \( \perp \) \}
28. pbarrier (newcurr, newnd, *opInfo) // info for current attempt
29. pwb (\( RD_q \)); psync
30. if \( curr \rightarrow key = key \) then return false
31. Help(opInfo)
32. if opInfo \( \rightarrow \) result \( \neq \perp \) then return opInfo \( \rightarrow \) result
33. opInfo := new Info ()
34. return \( \langle pred, curr, predInfo, currInfo \rangle \)

**Procedure** (Node*, Node*, Info*, Info*) SEARCH (T key)

35. Node *pred, *curr
36. Info *predInfo, *currInfo
37. curr := head
38. currInfo := head \( \rightarrow \) info
39. while curr \( \rightarrow \) key < key do
40. pred := curr
41. predInfo := currInfo
42. curr := curr \( \rightarrow \) next
43. currInfo := curr \( \rightarrow \) info
44. return \( \langle pred, curr, predInfo, currInfo \rangle \)

5 Evaluation

5.1 Evaluated Implementations

For our experiments, we use the Harris’ ordered linked list [29, 30] as our example data structure. We compare our general approach (described in Algorithms 1 and 2) with capsules [6]. Capsules partition their code into capsules, each containing a single CAS operation, and replace each
Algorithm 4: Recoverable Linked List: DELETE and FIND (code for thread \( q \))

Procedure boolean DELETE (T \( \text{key} \))

45 Info *\( \text{opInfo} \) := new Info()
46 \( RD_q := \text{null} \)
47 \( \text{pbarrier}(RD_q) \)
48 \( \text{CP}_q := 1 \)  \( \text{// check-point; } RD_q \text{ is initialized} \)
49 \( \text{pwb(CP}_q) ; \text{psync} \)
50 while true do
51 Gather Phase  \( \text{// search for node to delete} \)
52 \( \langle \text{pred}, \text{curr}, \text{predInfo}, \text{currInfo} \rangle := \text{Search} (\text{key}) \)
53 if \( \text{curr} \rightarrow \text{key} \neq \text{key} \) then
54 \( \text{AffectSet} := \{ \langle \text{curr}, \text{currInfo} \rangle \} \)
55 else \( \text{AffectSet} := \{ \langle \text{pred}, \text{predInfo} \rangle, \langle \text{curr}, \text{currInfo} \rangle \} \)
56 Helping Phase  \( \text{// help other if necessary} \)
57 if isTagged(predInfo) then
58 \( \text{Help(predInfo)} \)
59 continue
60 else if isTagged(currInfo) then
61 \( \text{Help(currInfo)} \)
62 continue
63 if \( \text{curr} \rightarrow \text{key} \neq \text{key} \) then
64 \( \text{WriteSet} := \emptyset \)
65 \( \text{opInfo} \rightarrow \text{result} := \text{false} \)
66 else
67 \( \text{WriteSet} := \{ \langle \text{pred} \rightarrow \text{next}, \text{curr} \rightarrow \text{next} \rangle \} \)
68 \( *\text{opInfo} := \{ \langle \text{DELETE}, \text{AffectSet}, \text{WriteSet}, \emptyset, \bot \rangle \} \)
69 \( \text{pbarrier}(*\text{opInfo}) \)
70 \( RD_q := \text{opInfo} \)  \( \text{// info for current attempt} \)
71 \( \text{pwb}(RD_q) ; \text{psync} \)
72 if \( \text{curr} \rightarrow \text{key} \neq \text{key} \) then return false
73 \( \text{Help(opInfo, true)} \)
74 if \( \text{opInfo} \rightarrow \text{result} \neq \bot \) then return \( \text{opInfo} \rightarrow \text{result} \)
75 \( \text{opInfo} := \text{new Info()} \)

Procedure boolean FIND (T \( \text{key} \))

76 Info *\( \text{opInfo} \) := new Info()
77 while true do
78 Gather Phase
79 \( \langle -, \text{curr}, -, \text{currInfo} \rangle := \text{Search}(\text{key}) \)
80 \( \text{AffectSet} := \{ \langle \text{curr}, \text{currInfo} \rangle \} \)
81 Helping Phase
82 if isTagged(currInfo) then
83 \( \text{Help(currInfo)} \)
84 continue
85 \( \text{result} := \langle \text{curr} \rightarrow \text{key} = \text{key} \rangle \)
86 \( \text{opInfo} \rightarrow \text{result} := \text{result} \)
87 \( \text{pbarrier}(*\text{opInfo}) \)
88 \( RD_q := \text{opInfo} \)
89 \( \text{pwb}(RD_q) ; \text{psync} \)
90 return result

CAS with a recoverable version of it \[3\]. In general, a single operation may be partitioned to multiple capsules, but for the restricted case of a normalized implementation \[45\], capsules can be optimized so that each operation is partitioned to only two capsules. In our experiments, the normalized variant consistently outperformed the general variant, so we only present the results of the former. We use the term capsules to refer to its normalized implementation below.
Determining the capsules boundaries can be done automatically [6]. However, to appropriately add persistence instructions to capsules without jeopardizing their applicability, it is proposed in [6] to use a general durability transformation [32] (which adds \texttt{pwb} and \texttt{pfence} after each access to shared memory). As our experiments show this results in prohibitive cost.

We compare the detectably recoverable linked list implementation of Section 4, which we call \textsc{Tracking}, with a linked list implementation, called \textsc{Capsules}, we implemented by applying the \textit{capsules} transformation (plus the durability transformation of [32]) to Harris’ ordered linked list [29, 30]. We also compare with Romulus [15], a detectably recoverable transactional memory system (which is blocking). We also measured the performance of CX-PUC [16], CX-PTM [16], and the Redo family of algorithms (i.e., Redo, RedoTimed, RedoOpt), presented in [16], as well as that of OneFile [41] (which are wait-free). RedoOpt constantly outperformed OneFile and all other algorithms in [16], so we present the diagrams only for RedoOpt, \textsc{Capsules}, and Romulus, below.

We have also undertaken the challenging task of adding persistence instructions in a manual, hand-tuned way to the \textsc{Capsules} implementation we produced. This resulted in \textsc{Capsules-Opt}, in which we avoid to persist most accesses to shared memory while traversing the list. Specifically, in \textsc{Capsules-Opt}, a thread executing an operation \textit{Op} with parameter \textit{k}, persists only the marked nodes it visits during \textit{Op}’s execution, as well as the nodes in the neighborhood of \textit{Op}’s target node, i.e., the two nodes preceding the first node of the list containing a key equal to or greater than \textit{k}. If a node is \textit{logically} deleted (i.e., if it is marked), all threads traversing it must persist it. Otherwise, the following bad scenario may happen: a thread executing \textit{Find}, searching for a node containing a key \textit{k}, which has been logically deleted without persisting its marked bit, may run to completion and return \texttt{false}. Then, a crash may cause the logically deleted node to appear in the linked list as unmarked. A subsequent \textit{Find} would then return \texttt{true}, which is incorrect. Persisting the nodes in the neighborhood of an operation’s target node is also necessary to avoid similar inconsistencies. We did not experiment with tree-like (or other) data structures, as that would require to produce \textsc{Capsules}-based implementations for them, which is a highly challenging task.

5.2 Experimental setting and benchmarks

We used a 48-core machine with 2 Intel Xeon Platinum 8260M 2.40GHz CPUs, with 24 cores each, and each core executing two hardware threads concurrently (for a total of 96 hardware threads). Our machine is equipped with a 1TB Intel Optane DC persistent memory (DCPMM) and the system is configured in AppDirect mode. We use the 1.9.2 version of the \textit{Persistent Memory Development Kit} [40], which provides the \texttt{pwb} and \texttt{psync} persistence instructions. We implement a \texttt{pfence} using a \texttt{psync}, since our machine does not support a \texttt{pfence} instruction. The machine runs Linux with kernel version 3.4. Code is written in C++ and compiled using g++ (version 4.8.5) with O3 optimizations. Each experiment lasts 10 seconds and each data point is the average of 10 experiments.

For the experiments, keys are chosen uniformly at random from the range [1, 500], [1, 1000], [1, 1500], and [1, 2000]. The list is initially populated by performing 250, 500, 750, and 1000 \texttt{Insert}s of random keys, respectively, resulting in an almost 40%-full list. We present update-intensive (30% finds) and read-intensive (70% finds) benchmarks. Results for other operation type distributions were similar.

5.3 Experimental Analysis

We start with our results for key range [1, 500]; as shown below, experiments for other ranges exhibit the same trends as the diagrams described here. Throughput evaluation results are shown in Figures 3a and 3b. The throughput of \textsc{Capsules} is extremely low due to the overhead imposed by applying the transformation in [32]. \textsc{Tracking} exhibits much better performance.
Figure 3: Throughput, number of \texttt{psync}s, throughput without \texttt{psync}s, number of \texttt{pwb}s, categorization of \texttt{pwb}s, and throughput of \texttt{pwb} categories, for evaluated implementations, with keys in the range $[1, 500]$ for read-intensive benchmark.

Figure 4: Throughput, number of \texttt{psync}s, throughput without \texttt{psync}s, number of \texttt{pwb}s, categorization of \texttt{pwb}s, and combined impact of \texttt{pwb} categories, for evaluated implementations with keys in the range $[1, 500]$ for update-intensive benchmark.

(Despite that, for preserving generality, we did not perform any hand-tuning to optimize its performance).

We next compare Tracking with Capsules-Opt. Recall that Capsules-Opt has been optimized in a hand-tuned manner, whereas we did not apply any hand-tuned persistence optimization to Tracking. Figures 3a and 4a show that Tracking has better performance.
than CAPSULES-OPT when the number of threads is large. Diagrams TRACKING[no pwbs] and CAPSULES-OPT[no pwbs] in Figures 3f and 4f show the performance of the algorithms when persistence instructions are excluded from their code. The performance of CAPSULES-OPT is better than that of TRACKING, in the absence of persistence instructions. It follows that the persistence cost of TRACKING is lower than that of CAPSULES-OPT.

We conducted a comprehensive experimental analysis to understand in detail the persistence overhead of the two algorithms. Figures 3g and 4g show that TRACKING performs more psync instructions than CAPSULES-OPT. To measure the actual overhead of these psync instructions, we removed psync (and thus also pfence) instructions from the code of both algorithms (Figures 3c and 4c). Despite the large attention that previous work [47, 14, 15, 24, 16] has paid on reducing the number of psync instructions that are incurred by recoverable implementations, Figures 3c and 4c show that this overhead is negligible. Specifically, the red and purple diagrams for TRACKING are almost identical, and the same is true for the blue and gray diagrams of CAPSULES-OPT. These results show that it is not the number of psync instructions that greatly affects the performance of the tested algorithms. The reason for this is that a CAS on an Intel Xeon machine serializes all outstanding store operations (that is, it waits for them to complete) [31, Section 8.1.2.2]. Thus, a CAS behaves like if it executes an sfence. Since a psync instruction is implemented using sfence in our Intel machine, this has as a result, many psync instructions to be applied on empty (or nearly empty) store buffers, thus incurring negligible performance cost.

We next focus on the pwb instructions, as it should be those that cause the better performance of TRACKING over CAPSULES-OPT. Counter-intuitively, Figures 3d and 4d show that TRACKING performs a larger number of pwb instructions than CAPSULES-OPT, for all the tested benchmarks. So, we decided to conduct additional experiments to measure the overhead of each single pwb instruction. To do so, we removed all code lines containing persistence instructions from each persistent implementation to get its persistence-free version, and then we measured the impact of adding each of the pwb code lines in the persistence-free version. For simplicity of presentation, we categorize the code lines containing pwb instructions into three categories according to their impact on performance, namely those with low, medium, or high performance impact. A pwb code line has low impact, if it results in at most 10% performance loss when inserting it. The insertion of a code line in the second category (medium impact) results in performance loss between 10% and 30%, whereas if we insert a code line of the third category (high impact), we will see more than 30% performance loss. Note that each code line may be executed many times in an execution and thus its performance impact expresses the total performance loss that the execution of all its instances cause.

Thus, we have three sets of code lines, namely L, M, and H, containing the code lines that have low, medium, and high performance impact, respectively. Figures 3e and 4e show that TRACKING mostly performs low-cost pwbs. Just a few of them are of medium cost, whereas no pwbs belong in the high-cost category. In contrast, almost 50% (and in some cases up to 70%) of the executed pwbs in CAPSULES-OPT are of high cost, and the rest are mainly of low cost and some of them (up to almost 10%) are of medium cost. Remarkably, TRACKING performs at least four low performance impact pwbs per update operation (to persist the values of variables CP and RD, as well as the newly allocated data). This increases the number of pwbs that TRACKING executes, without however resulting in high performance overheads.

By inspection of the algorithms’ codes, we observed that a low-cost pwb is applied either on a private non-volatile variable used by a thread to track its own progress or on newly-allocated data that has not yet become shared. We also observed that a pwb that incurs high performance penalty is executed on a shared variable (cache line) that is accessed by many other threads, as such pwbs will result in a high number of cache misses (and in increased traffic on the memory controller). In particular, when pwbs precede a CAS instruction on the same variable (or more generally, on the same cache line), the execution time of both the pwb and the CAS can be
increased. This is because the same cache line may have to be moved between the cache and the NVMM multiple times (depending on the degree of contention). Specifically, the \textit{CAS} will wait for outstanding store operations to complete, which will cause the cache line affected by the \texttt{pwb} to be flushed and invalidated in cache. Then, this cache line is re-fetched for executing the actual update of the \textit{CAS}, resulting in performance overhead. In case many threads issue first a \texttt{pwb} instruction and then a \textit{CAS}, on the same cache line, the above scenario will occur repeatedly. Similar performance overheads may arise when \texttt{pwb}s are executed (by different threads) after a \textit{CAS} (on the same cache line), as these \texttt{pwb}s will cause cache misses (and increased traffic on the memory controller). A \texttt{psync} (or \texttt{pfence}) instruction following \texttt{pwb}s executed by different threads on the same cache line will also cause performance overhead.

Let $X$ be any of the three sets, $L$, $M$, or $H$. We say that the $X$-caused performance loss is the performance loss we see when all code lines in $X$ are added in the persistence-free version. Figures 5 and 6 shows the $X$-caused performance loss for \textsc{Tracking} and \textsc{Capsules-Opt}, respectively, for all three values of $X$. These experiments reveal that the $X$-caused performance loss is at least as high as the impact of each code line of category $X$. However, the $X$-caused performance loss is not necessarily the sum of the impacts of the code lines in $X$. Depending on contention, it may be higher or lower than this sum. Thus, it is not enough to measure just the impact of each single \texttt{pwb} code line. Experiments to figure out their combined impact are also needed.

Figures 3f and 4f illustrate this combined impact. In these figures, we start from the original persistent algorithm, and remove one by one the different categories of \texttt{pwb}s, starting from $L$, continuing with $M$, and finally removing $H$, studying the increase in performance that the removal of each category causes. Not surprisingly, the diagrams show that the removal of the \texttt{pwb}s of category $L$ (low performance impact) do not have any significant impact, since the total persistence cost is dominated by the cost incurred by the \texttt{pwb}s of the other categories. Moreover, we observe that the removal of the \texttt{pwb}s of the category $H$ has the biggest impact in the performance of \textsc{Capsules-Opt}. Thus, the diagrams show that \textsc{Tracking} owes its good performance to the fact that it executes just a few medium-cost \texttt{pwb}s. On the contrary, \textsc{Capsules-Opt} performs a lot of high-cost \texttt{pwb}s. The cost of the \texttt{pwb}s of category $M$ is the dominant persistence cost in \textsc{Tracking}. \textsc{Capsules-Opt} performs less \texttt{pwb}s of this category and their combined impact is not high.

We now present our results for other key ranges. Figure 7 illustrates the throughput of all linked list algorithms for the following key ranges: $[1,1000]$, $[1,1500]$, and $[1,2000]$. Results for the read-intensive workload are presented on the left and those for the update-intensive workload are presented on the right. Figures 8 and 9 show the numbers of \texttt{pwb} instructions,
Figure 6: Impact of \textit{pwb} categories on performance of \textsc{Capsules-Opt}.

Figure 7: Throughput for key ranges $[1, 1000]$, $[1, 1500]$, and $[1, 2000]$ (the read-intensive benchmarks are shown on top and the update-intensive benchmarks are shown on bottom).

their categorization, and the combined impact of these categories, for the read-intensive and the update-intensive benchmarks, respectively, and key ranges: $[1,1000]$, $[1,1500]$, $[1,2000]$. Figures 10 and 11 show the performance impact when the \textit{pwb}s of each category are added in the persistence-free version of \textsc{Tracking} and \textsc{Capsules-OPT}, respectively, for key ranges: $[1,1000]$, $[1,1500]$, $[1,2000]$. These figures show the same trends exhibited by the experiments presented above for key range $[1,500]$.

Summarizing, our results suggest that in a machine with Intel Optane, measuring only the number of \textit{pwb}s is not enough to fully understand the persistence cost of a recoverable algorithm. A thorough evaluation is required since the impact of different \textit{pwb}s on performance may greatly vary. Specifically, a categorization of the persistence instructions may be necessary to better understand their impact on performance. This categorization, together with experiments like those presented in Figures 5 and 6, reveal useful information about the persistence cost of an algorithm. Thus, they may provide good insights to algorithms’ designers for improving the persistence cost of their algorithms. Additionally, experiments like those presented in Figures 3 and 5 reveal useful information about the persistence cost of an algorithm.
Figure 8: Number of \texttt{pwb}s (top), categorization of \texttt{pwb}s (middle), and combined impact of \texttt{pwb} categories (bottom), for evaluated implementations with keys in the ranges $[1, 1000]$, $[1, 1500]$, $[1, 2000]$ for read-intensive benchmark.

and [4] are more useful for performing an in depth comparison of the persistence cost of different algorithms.

Although in the algorithms we study, the \texttt{psync} instructions do not have any significant impact in performance, to fully understand the performance of other recoverable algorithms, a thorough evaluation may be required for \texttt{psync} as well (recall e.g., that a \texttt{psync} instruction following \texttt{pwb}s executed by different threads on the same cache line may cause performance overhead).

6 Detectably Recoverable Versions of Additional Data Structures

We briefly discuss additional data structures that can become detectably recoverable by applying the Tracking approach.

6.1 Detectably Recoverable Binary Search Tree

The algorithm in [19] (LF-BST) implements a \textit{leaf-oriented} (external) binary search tree. It uses \texttt{CAS} to flag an internal node whenever a child pointer of it is to be changed, and to mark it whenever it is to be deleted. A thread $p$, executing an update $Op$, allocates a descriptor where it records the information needed by other threads to help $Op$ complete. Each internal
Figure 9: Number of pwb (top), categorization of pwb (middle), and combined impact of pwb categories (bottom), for evaluated implementations with keys in the ranges [1, 1000], [1, 1500], [1, 2000] for update-intensive benchmark.

Figure 10: Impact of pwb categories on performance of Tracking, with key ranges [1,1000], [1,1500], and [1,2000], for both read-intensive (top) and update-intensive (bottom) benchmarks.
Figure 11: Impact of pwb categories on performance of CAPSULES-OPT, with key ranges [1,1000], [1,1500], and [1,2000], for both read-intensive (top) and update-intensive (bottom) benchmarks.

\[
\text{type } \text{Internal} \{ \\
\quad \triangleright \text{subtype of Node} \\
\quad \text{Key} \cup \{\infty_1, \infty_2\} \text{ key} \\
\quad \text{Node} *\text{left}, *\text{right} \\
\quad \text{Info} *\text{info} \\
\}
\]

\[
\text{type } \text{Leaf} \{ \\
\quad \triangleright \text{subtype of Node} \\
\quad \text{Key} \cup \{\infty_1, \infty_2\} \text{ key} \\
\}
\]

\[
\triangleright \text{Initialization:} \\
\text{Internal} *\text{Root} := \text{pointer to new Internal node with key field } \infty_2, \\
\text{info field NULL, and pointers to new Leaf nodes with keys } \infty_1 \text{ and } \infty_2, \text{ respectively, as left and right fields.} \\
\]

\[
\triangleright \text{Assume the set contains two special values, } \infty_1 < \infty_2, \text{ such that every other key } k, \infty_1 < k < \infty_2
\]

Figure 12: BST type definitions and initialization.

node contains an update field which stores a reference to a descriptor and a 2-bits status field which indicates whether the node is flagged for insertion, flagged for deletion, marked, or clean. Each successful flag or mark CAS installs a pointer to the descriptor of the relevant operation in the update field of the node it is applied on.

We employ Tracking (Algorithms 1 and 2) to make LF-BST detectably recoverable. The data types, shared variables, and initialization values of the detectably recoverable binary search tree implementation appear in Figure 12. Algorithms 5 and 6 present the code for INSERT, DELETE and FIND.

Consider an operation Op and let \( l, p \) and \( gp \) be pointers to the leaf Op’s search arrives at, to its parent and to its grandparent, respectively. If Op is an INSERT, it replaces the node pointed to by \( l \) with a subtree of three nodes. Thus, Op’s AffectSet contains a pointer to \( l \) and a pointer to \( p \) (as its child pointer will change to point from \( l \) to the root of the new subtree). Op’s WriteSet contains \( p \), and Op’s NewSet contains the three new nodes of the subtree that replaces \( l \). If Op is a DELETE, Op changes the appropriate child pointer of \( gp \) to point to the sibling of \( l \). For applying Tracking, we need to create a copy of this sibling, to avoid the ABA
Algorithm 5: Recoverable BST: INSERT and SEARCH

**Procedure** boolean INSERT (Key k)

1. Leaf *new := new Leaf node whose key field is k
2. Info *opInfo := new Info ()
3. RDq := ⊥
4. pbARRIER (RDq) // check-point; RDq is initialized
5. CPq := 1 // check-point; RDq is initialized
6. pwb (CPq); psync
7. while true do
   8.   Gather Phase // search for location to insert
      9.     ⟨−, p, l, −, pInfo⟩ := Search(k)
     10.     AffectSet := {(p, pInfo)}
8.   Helping Phase // help other if necessary
9.   if isTagged(pInfo) then
10.      Help(pInfo); continue
11.     newSibling := a new Leaf whose key is l.key // make a duplicate of l
12.     newInternal := a new Internal node with key field max(k, l.key), and with two child fields
     13.      equal to new and newSibling (the one with the smaller key is the left child)
14.     if l → left then
15.        WriteSet := {(p → left, l, newInternal)}
16.     else WriteSet := {(p → right, l, newInternal}  
17.     NewSet := {newInternal}
18.     *opInfo := pointer to a new operation descriptor { INSERT, AffectSet, WriteSet, NewSet, ⊥}
19.     newInternal → info := getTagged(opInfo)
20.     if l → key = k then return false // key k is in the tree
21.     opInfo → result := false  
22.     pbARRIER (newSibling, newInternal, *opInfo)
23.     RDq := opInfo // info for current attempt
24.     pwb (RDq); psync
25.     if l → key = k then return false
26.     Help(opInfo, true)
27.     if opInfo → result ≠ ⊥ then return opInfo → result

**Procedure** (Internal*, Internal*, Leaf*, Info*, Info*) SEARCH (Key k)

▷ Used by INSERT, DELETE and FIND to traverse a branch of the BST
30. Internal *gp, *p
31. Node *l := Root
32. Info *gpInfo, *pInfo
33. while l points to an internal node do
34.     gp := p // remember grandparent
35.     p := l // remember parent
36.     gpInfo := pInfo // remember info field of gp
37.     pInfo := p → info // remember info field of p
38.     if k < l → key then l := p → left else l := p → right // move down to appropriate child
39. return ⟨gp, p, l, gpInfo, pInfo⟩

problem. Therefore, Op’s AffectSet contains l, p, gp, and a pointer to l’s sibling; its WriteSet contains gp and its NewSet contains the new node that replaces l’s sibling. The AffectSet of a FIND contains only l. FINDs can be further optimized to have their AffectSet be equal to the empty set.

Threads can use the update field that already exists in the tree nodes and the descriptors used in LF-BST, to implement Tracking without any significant memory overhead. Also, the tagging mechanism is provided for free through the flagging and marking mechanisms of LF-BST.
Algorithm 6: Recoverable BST: DELETE and FIND.

Procedure boolean DELETE (Key k)
40 Info *opInfo := new Info ()
41 RDq := ⊥
42 pbarrier (RDq) // check-point; RDq is initialized
43 CPq := 1
44 pwb (CPq); psync
45 while true do
46 | Gather Phase // search for node to delete
47 | ⟨gp, p, l, gpInfo, pInfo⟩ := Search(k)
48 | AffectSet := {(gp,gpInfo),(p,pInfo)}
49 | Helping Phase // help other if necessary
50 | if isTagged(gpInfo) then
51 | | Help (gpInfo); continue
52 | if isTagged(pInfo) then
53 | | Help (pInfo); continue
54 | if l = p → left then other := p → right
55 | else other := p → left
56 | if p = gp → left then
57 | | WriteSet := {(gp → left,p, other)}
58 | else WriteSet := {(gp → right,p, other)}
59 | *opInfo := pointer to a new operation descriptor (DELETE, AffectSet, WriteSet, ⊥, ⊥)
60 | if l → key ≠ k then
61 | | opInfo → result := false
62 | pbarrier (*opInfo)
63 | RDq := opInfo // info for current attempt
64 | pwb (RDq); psync
65 | if l → key ≠ k then return false
66 | Help(opInfo, true)
67 | if opInfo → result ≠ ⊥ then return opInfo → result

Procedure boolean FIND (Key k)
68 Info *opInfo := new Info ()
69 while true do
70 | Gather Phase
71 | ⟨−, p, l, −, pInfo⟩ := Search(k)
72 | AffectSet := {(p,pInfo)}
73 | Helping Phase
74 | if IsTagged(pInfo) then
75 | | Help(pInfo)
76 | continue
77 result := (l → key = k)
78 opInfo → result := result
79 pbarrier (*opInfo)
80 RDq := opInfo
81 pwb (RDq); psync
82 return result

6.2 Detectably Recoverable Exchanger

An Exchanger [30, 43] allows two threads to pair-up their operations and exchange values. The first thread, p, that arrives to an Exchanger, finds it free and captures it by atomically writing to it its value. Then, p busy-waits until another thread q collides with it: if q arrives while p is waiting, it reads p’s value in the Exchanger, and tries to atomically write its value to it and inform p of a successful collision.

We employ Tracking to achieve recoverability: we implement the exchanger as a pointer,
Algorithm 7: Recoverable Exchanger

Type Node {
        {Waiting, Busy} state
        T val
        Info *ini
    }

Shared variables:
    Node *slot := pointer to a new Node with fields
    (Busy, ⊥, Null)

Private variables:
    Info *RDq for each process q initially ⊥

Procedure T Exchange(Node *slot, T value)
    Node *newNd := new Node with val field
    value
    Info *opInfo := new Info ()
    pbarrier (RDq)
    CPq := 1 // check-point; RDq is initialized
    pwb (CPq); psync
    while true do
        begin Gather Phase
            curr := slot
            currInfo := curr → info
            AffectSet := {curr, currInfo}
        end
        begin Helping Phase
            if isTagged(currInfo) then
                Help(currInfo); continue
            end
            if slot → state = Waiting then
                newNd → state := Busy
                WriteSet := { (slot, curr, newNd),
                        (currInfo → result, empty, value ),
                        (opInfo → result, ⊥, curr → val ) }
            else
                newNd → state := Waiting
                WriteSet := { (slot, curr, newNd), (opInfo
                        → result, ⊥, empty) }  // failed
            end
            newNd → info := getTagged(opInfo)
            NewSet := { newNd }
            opInfo := { Exchange, AffectSet, WriteSet, NewSet, ⊥}
            pbarrier (newNd, *opInfo)
            RDq := opInfo // info for current attempt
            pwb (RDq); psync
            Help(opInfo)
        end
        if opInfo → result ≠ ⊥ then
            if newNd → state = Waiting then
                while slot = newNd do;
                sInfo := slot → info
                if isTagged(sInfo) then Help(sInfo)
            end
            return opInfo → result
        end
        opInfo := new Info ()
    end

Procedure Help (Info *opInfo)

begin Tagging Phase
    (curr, currInfo) := first element in
    opInfo → AffectSet
    result := CAS(curr → info, currInfo,
    getTagged(opInfo)) // tag CAS
    pub (curr → info)
    if result ≠ currInfo and result ≠
    getTagged(opInfo) then // curr is not tagged
        psync ()
    return // operation attempt failed
    psync ()
end

begin Update Phase
    foreach (ptr, oldVal, newVal) in WriteSet do
        CAS(ptr, oldVal, newVal)
        pub (ptr)
        psync ()
    end

begin Cleanup Phase
    foreach node nd in (AffectSet ∪ NewSet) do
        CAS(nd → info, getTagged(opInfo),
        getUnTagged(opInfo))
        pub (nd → info)
        psync
    end
    return
end

Procedure boolean

Exchange-Recover (Node *slot, T value)

begin
    Info *opInfo := RDq
    if CPq = 0 or opInfo = ⊥ then
        Re-invoke Exchange
        Help(opInfo)
    if opInfo → result ≠ ⊥ then // operation completed
        newNd := node in opInfo → NewSet
        if newNd → state = Waiting then
            while slot = newNd do;
            sInfo := slot → info
            if isTagged(sInfo) then Help(sInfo)
        end
        return opInfo → result
        else Re-invoke Exchange // operation attempt failed
end

slot, that points to a node. This node stores the current state of the exchanger, the value of the last thread accessing it and a pointer to a descriptor. Every time a thread p wants to initiate an exchange, it allocates a new node nd’ containing its value, and access the node nd’ pointed to by slot to find the current state of the exchanger. If nd’ is not tagged, then p attempts to install its own descriptor into it. Otherwise, another thread q has already attempted to perform an exchange, so p has to help q to finish the exchange. Algorithm 7 presents the code for this detectably recoverable exchanger implementation.
7 Related Work and Discussion

We present the Tracking approach for detectable recovery of concurrent data structures and apply it to several well-known concurrent data structures. Our approach is general and it yields recoverable implementations from their non-recoverable counterparts, preserving their efficiency. Specific recoverable concurrent implementations of data structures were presented, such as mutual exclusion locks [27, 28], stacks and queues [25, 34, 44, 20, 21, 42], heaps [21], hash maps (see e.g., [36, 52]), and B-trees (see e.g., [11, 36, 37]), with optimizations exploiting specific aspects of the objects.

Tracking ensures *nesting-safe recoverable linearizability* (NRL) [3] and *durable linearizability* (DL) [32]. Operation descriptors were used in DL implementations of several data structures [12, 39, 49], and other transformations that avoid logging [17, 32], but none of these ensures detectability.

The recoverable log queue [25] augments queue nodes with tracking information, which is used after a system-wide crash to *synchronously* try and complete all pending operations from the previous phase before starting a new phase. Other recoverable queues [25, 44] are not detectable.

Tracking shares some similarities with SiloR [51], but there are several important differences. Tracking does not maintain a history of records, whereas SiloR maintains such a history starting from the last checkpoint. In Tracking, each thread logs a record about its last operation only and these records are used for ensuring not only persistence but also lock-freedom. They may be installed into nodes by any active thread, so there is no need for designated threads (as is the case in SiloR). Finally, Tracking proposes an efficient scheme for flushing writes to byte-addressable NVM, which is different from optimizing block-writes to disk.

We present a methodology for in-depth understanding of the (individual and combined) cost incurred by persistence instructions. Some of our observations were also made in [44], e.g., that accesses to flushed content are of high cost. Our experimental analysis enriches these observations by providing new insights, which lead to a detailed scheme for measuring the persistence cost of recoverable algorithms.

An NRL implementation can be obtained from any algorithm using only read, write and CAS primitives by replacing each primitive with its (NRL) recoverable version (see [3]). Implementations using only read and CAS can be made recoverable and detectable using capsules [6] (see Section 5).

A recent paper [26] follows a similar approach in order to transform any lock-free implementation into a recoverable one that satisfies durable linearizability. Another recent work [24] presents a transformation of lock-free implementations to their durable linearizable versions; the transformation can be applied on a specific class of concurrent data structures, called *traversal* data structures. These transformations, published after the preliminary version of our work [1, 2], yield non-detectable implementations.

A recoverable lock-free universal implementation [14] requires only one round trip to NVMM per operation, which is optimal. This construction (as well as [24, 26]) makes the strong assumption that a single recovery function is executed upon recovery, consistently reconstructing the data structure, whereas we allow failed threads to be recovered by the system in an asynchronous manner. Other logging-based approaches are [9, 13, 35, 50].

Romulus [15] is a transactional memory algorithm that provides durability and detectability. However, it is blocking, satisfying only starvation-freedom for update transactions.

Our recoverable implementations—as well as the original, non-recoverable implementations—rely on garbage collectors that correctly recycle memory once it becomes unreachable. This naturally motivates the question of implementing lock-free recoverable memory managers [7, 41],

24
which we hope to investigate in future work.

Acknowledgments. This work was supported by the EU Horizon 2020, Marie Skłodowska-Curie project with GA No 101031688 and by the ISF grant 380/18. It was also co-financed by Greece and the European Union (European Social Fund- ESF) through the Operational Programme “Human Resources Development, Education and Lifelong Learning” in the context of the project “Reinforcement of Postdoctoral Researchers - 2nd Cycle” (MIS-5033021), implemented by the State Scholarships Foundation (IKY).

References


[24] M. Friedman, N. Ben-David, Y. Wei, G. E. Blelloch, and E. Petrank. Nvtraverse: In nvram data structures, the destination is more important than the journey. In Proceedings


A Artifact

The artifact contains the source code and scripts to reproduce the experimental results of this paper. The code for our algorithms, together with additional recoverable implementations we have produced, can be found in the following GitHub repository:

https://github.com/ConcurrentDistributedLab/Tracking

A.1 Requirements

The following libraries are required in order to be able to compile and run our code:

1. A modern 64-bit multi-core machine supporting non-volatile main memory.
2. A recent Linux distribution.
3. The g++ (version 4.8.5 or greater) compiler.
4. Building requires the development versions of the following packages:
   - libatomic,
   - libnuma,
   - libvmem, necessary for building the persistent objects, and
   - libpmem, necessary for building the persistent objects.

   Depending on the directory in which these packages are installed, the appropriate environment variable (for instance, the LD_LIBRARY_PATH variable in Linux) should contain the path to them.

A.2 Reproduce experimental results

To compile the executables, the figures_compile.sh script should be executed. Then, to run the experiments and produce the results of each figure in Section 5 regarding our algorithms, the figures_run.sh should be executed; it creates the output files in the results directory. Finally, to plot the figures the figures_plot.py python script should be executed.

The folder Expected Results contains the expected results and figures for our algorithm (Tracking). After compiling the executables, a custom experiment can be run by calling:

./⟨executable⟩ ⟨algorithm⟩ [threads_number] [duration(seconds)]

A.3 License

This code is provided under the LGPL-2.1 License.