Detectable Recovery of Lock-Free Data Structures

Hagit Attiya*
Technion
Israel
hagit@cs.technion.ac.il

Ohad Ben-Baruch*
Ben-Gurion University
Israel
ohadben@post.bgu.ac.il

Panagiota Fatourou†
Université de Paris, LIPADE, F-75006
Paris, France
FORTH and University of Crete
Greece
faturu@csd.uoc.gr

Danny Hendler*
Ben-Gurion University
Israel
hendlerd@cs.bgu.ac.il

Eleftherios Kosmas‡
University of Crete, Computer
Science Department
Greece
ekosmas@csd.uoc.gr

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.

CCS Concepts: **• Theory of computation → Concurrent algorithms**; **• Computing methodologies → Concurrent algorithms**; **• Information systems → Data structures**; **• Hardware → Non-volatile memory**.

Keywords: non-volatile memory, NVM-based computing, persistence, recoverable algorithms and data structures, concurrent data structures, linked-list, tree, exchanger, synchronization, lock-freedom, persistence cost analysis

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

*Supported by ISF grant 380/18.
†Supported by the EU Horizon 2020, Marie Sklodowska-Curie project with GA No 101031688.
‡This research is 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).

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.

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9204-4/22/04…$15.00
https://doi.org/10.1145/3503221.3508444

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 different 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 (*pwb*)* instruction; a pwb flushes all fields fitting in a cache line. The order of pwbs is not necessarily preserved. A pfence instruction orders preceding pwbs before all subsequent pwbs. A psync instruction waits until all previous pwbs 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.[1] 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.

[1] System support is necessary for designing detectable algorithms [5].

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 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

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
           then
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
         then
18         opInfo → result := response determined based on
             opInfo
19     pbarrier (*opInfo, NewSet)
20     RD_q := opInfo
21     pwb (RD_q); psync
22     if WriteSet = ∅ and AffectSet contains only one element
         then
23         return opInfo→result
24     Help(opInfo)
25     if opInfo→result ≠ ⊥ then return opInfo→result
26     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
```

Algorithm 2: Tracking - Help (code for thread q)

```
Procedure Help (Info *opInfo)
32 Tagging Phase
33     ⟨nd, ndinfo⟩ := head of opInfo→AffectSet
34     while nd ≠ Null do
35         res := CAS(nd→info, ndinfo, getTagged(opInfo))
36         pwb (nd→info)
37         if res ≠ ndinfo and res ≠ getTagged(opInfo) then
38             Backtrack Phase
39                 ⟨nd, ndinfo⟩ := previous element in
                     AffectSet
40                 while nd ≠ Null do
41                     CAS(nd→info, getTagged(opInfo),
                         getUntagged(opInfo))
42                     pwb (nd→info)
43                     ⟨nd, ndinfo⟩ := previous element in
                         AffectSet
44                 psync
45                 return
46         ⟨nd, ndinfo⟩ := next element in AffectSet
47     psync
48 Update Phase
49     foreach Update structure st in WriteSet do
50         perform the update by executing CAS based on
             information contained in st
51         pwb (updated field)
52 opInfo→result := response of the operation described by
     opInfo
53 pwb (opInfo→result); psync
54 Cleanup Phase
55     foreach node nd in (AffectSet ∪ NewSet) which is still
         part of the data structure do
56         CAS(nd→info, getTagged(opInfo),
             getUntagged(opInfo))
57         pwb (nd→info)
58     psync
```

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 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.

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.

Op then proceeds to its helping phase. If an *info* field *ndInfo* (of a node *nd*) in *AffectSet* is tagged, then Help is used to complete the operation that tagged the node, before starting a new attempt. After the helping phase, the *WriteSet* and the *NewSet*—needed to complete *Op*—are created. The *WriteSet* 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`.

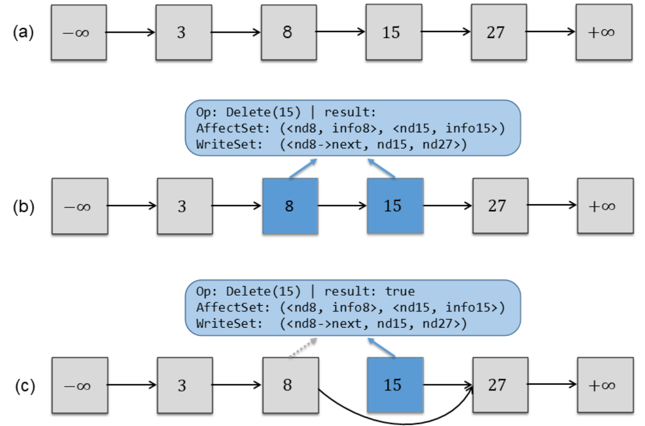


Figure 1. Example of how Tracking works on a sorted linked list.

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) 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., [8, 19, 22, 38]).

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

To support detectable recovery, a pointer to the descriptor used in each attempt is stored in RD_q before Help is called. If the check-point is not set or RD_q is still Null, *Op* has made no changes and can simply be restarted. Otherwise, the descriptor pointed to by RD_q, *opInfo*, indicates whether the last attempt of *Op* was successful, and if not, whether it crashed while making changes. Since Help is idempotent, recovery can call Help(*opInfo*) to complete *Op*, in case it is still in progress. When Help returns, either *Op* took effect and *result* stores its response, or it did not and *result* is ⊥;

in the latter case, *Op* can be re-invoked. It is necessary to call Help first, to deal with the case in which *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 *Op*. Linearizability and lock-freedom for Tracking are discussed in [2].

Persistence instructions. After setting the check-point, allocating a descriptor and storing a reference to it in RD_q, a `pwb` followed by a `psync` ensures that the data is accessible upon recovery. A `pbarrier` after initializing RD_q ensures that `pwb`s 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.

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. More details are provided in [2].

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 $-\infty$ and $+\infty$. A node *nd* may be tagged either for update (indicating its *next*

```
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.

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, 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.

5 Evaluation

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 *CAS* with a recoverable version of

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 := ⊥
5  pbarrier (RD_q)
6  CP_q := 1          // check-point; RD_q is initialized
7  pwb (CP_q); psync
8  while true do
9      Gather Phase // search for location to insert
10         ⟨pred, curr, predInfo, currInfo⟩ := SEARCH(key)
11         if curr→key = key then
12             AffectSet := {⟨curr,currInfo⟩}
13         else AffectSet:= {⟨pred,predInfo⟩, ⟨curr,currInfo⟩}
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 := ⟨curr → key, curr → next,
        getTagged(opInfo)⟩
20     newnd → info := getTagged(opInfo)
21     if curr → key = key then              // key in list
22         WriteSet:= NewSet:= ∅;
23         opInfo → result := false
24     else
25         WriteSet:= {⟨pred→next, curr, newnd⟩}
26         NewSet:= {newnd, newcurr}
27     *opInfo := ⟨INSERT, AffectSet, WriteSet, NewSet, ⊥⟩
28     pbarrier (newcurr, newnd, *opInfo)
29     RD_q := opInfo          // info for current attempt
30     pwb (RD_q); psync
31     if curr → key = key then return false
32     HELP(opInfo)
33     if opInfo → result ≠ ⊥ then return opInfo → result
34     opInfo := new Info ()
```

```
Procedure ⟨Node*, Node*, Info*, Info*⟩ SEARCH (T key)
35 Node *pred, *curr
36 Info *predInfo, *currInfo
37 curr := head
38 currInfo := head →info
39 while curr→key < key do
40     pred := curr
41     predInfo := currInfo
42     curr := curr → next
43     currInfo := curr → info
44 return ⟨pred, curr, predInfo, currInfo⟩
```

Algorithm 4: Recoverable Linked List: Delete and Find (code for thread q)

```
Procedure boolean DELETE (T key)
45 Info *opInfo := new Info ()
46 RD_q := NULL
47 pbarrier (RD_q)
48 CP_q := 1          // check-point; RD_q is initialized
49 pwb (CP_q); psync
50 while true do
51     Gather Phase      // search for node to delete
52         ⟨pred, curr, predInfo, currInfo⟩ := SEARCH(key)
53         if curr→key ≠ key then
54             AffectSet := {⟨curr,currInfo⟩}
55         else AffectSet:= {⟨pred,predInfo⟩, ⟨curr,currInfo⟩}
56     Helping Phase       // help other if necessary
57         if isTagged(predInfo) then
58             HELP(predInfo)
59             continue
60         else if isTagged(currInfo) then
61             HELP(currInfo)
62             continue
63     if curr → key ≠ key then
64         WriteSet:= ∅
65         opInfo → result := false
66     else
67         WriteSet:= {⟨pred→next, curr, curr → next ⟩}
68     *opInfo := ⟨DELETE, AffectSet, WriteSet, ∅, ⊥⟩
69     pbarrier (*opInfo)
70     RD_q := opInfo        // info for current attempt
71     pwb (RD_q); psync
72     if curr → key ≠ key then return false
73     HELP(opInfo, true)
74     if opInfo → result ≠ ⊥ then return opInfo → result
75     opInfo := new Info ()
```

```
Procedure boolean FIND (T key)
76 Info *opInfo := new Info ()
77 while true do
78     Gather Phase
79         ⟨−, curr, −, currInfo⟩ := SEARCH (key)
80         AffectSet:= {⟨curr,currInfo⟩}
81     Helping Phase
82         if IsTagged(currInfo) then
83             HELP(currInfo)
84             continue
85     result := (curr → key = key)
86     opInfo → result := result
87     pbarrier (*opInfo)
88     RD_q := opInfo
89     pwb (RD_q); psync
90     return result
```

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 pwb and 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 Tracking, with a linked list implementation, called Capsules, we implemented by applying the *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, Capsules, and Romulus, below.

We have also undertaken the challenging task of adding persistence instructions in a manual, hand-tuned way to the Capsules implementation we produced. This resulted in Capsules-Opt, in which we avoid to persist most accesses to shared memory while traversing the list. Specifically, in Capsules-Opt, a thread executing an operation *Op* with parameter *k*, persists only the marked nodes it visits during *Op*'s execution, as well as the nodes in the neighborhood of *Op*'s target node, i.e. the two nodes preceding the first node of the list containing a key equal to or greater than *k*. If a node is *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 Find, searching for a node containing a key *k*, which has been logically deleted without persisting its marked bit, may run to completion and return `false`. Then, a crash may cause the logically deleted node to appear in the linked list as unmarked. A subsequent Find would then return `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 Capsules-based implementations for them, which is a highly challenging task.

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 *Persistent Memory Development Kit* [40], which provides the pwb and psync persistence instructions. We implement a pfence using a psync, since our machine does not support a 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 $[1, 500]$. (Experiments for other ranges can be found in [2]; they exhibit the same trends as the diagrams here.) The list is initially populated by performing 250 Inserts of random keys, 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.

Experimental Analysis. Throughput evaluation results are shown in Figures 3a and 4a. The throughput of Capsules is extremely low due to the overhead imposed by applying the transformation in [32]. Tracking exhibits much better performance (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 3b and 4b 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 [14–16, 24, 47] 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

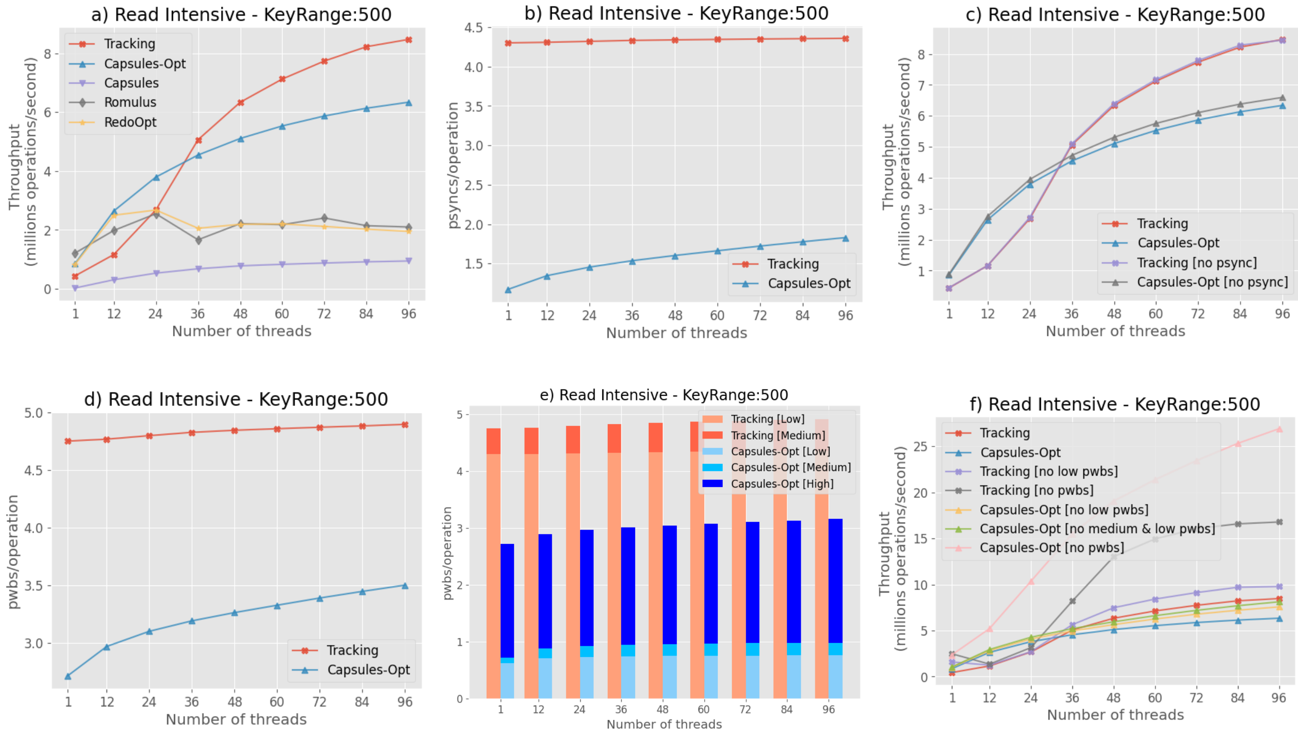


Figure 3. Throughput, number of psyncs, throughput without psyncs, number of pwbs, categorization of pwbs, and throughput of pwb categories, for evaluated implementations, with keys in the range $[1, 500]$ for read-intensive benchmark.

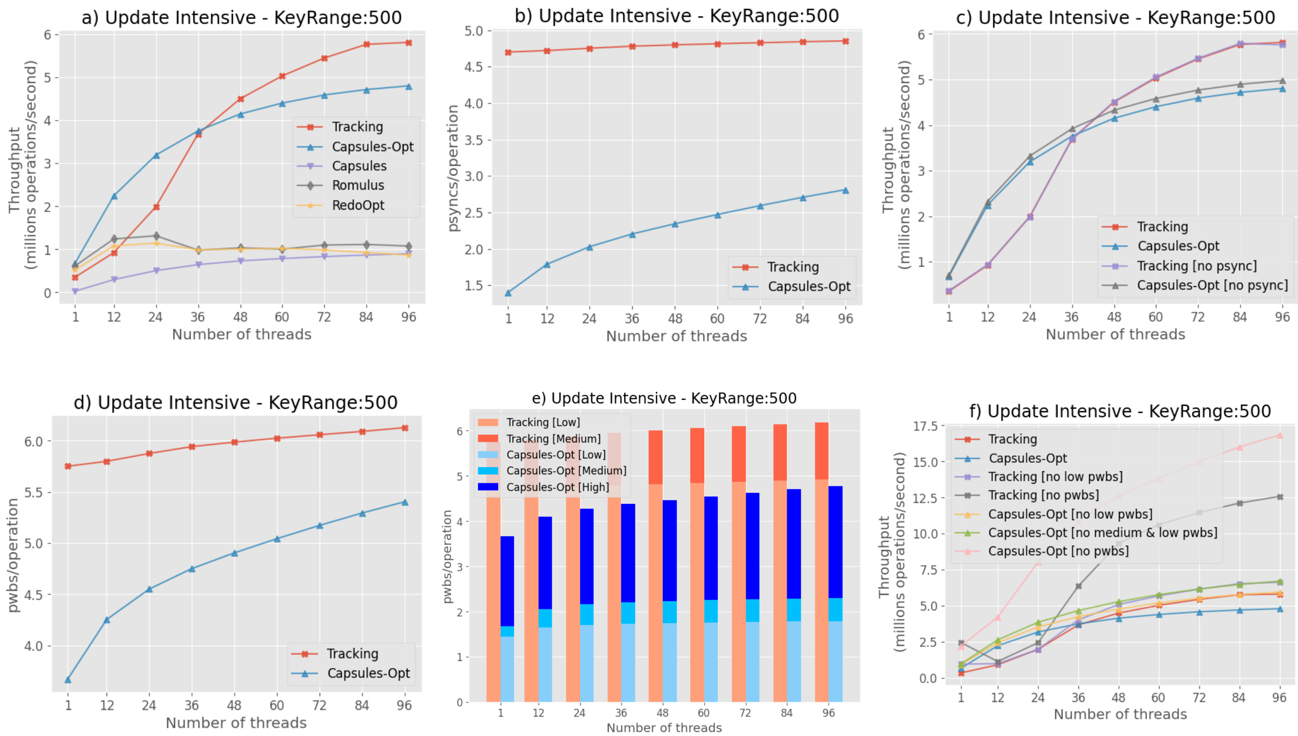


Figure 4. Throughput, number of psyncs, throughput without psyncs, number of pwbs, categorization of pwbs, and combined impact of pwb categories, for evaluated implementations with keys in the range $[1, 500]$ for update-intensive benchmark.

(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.

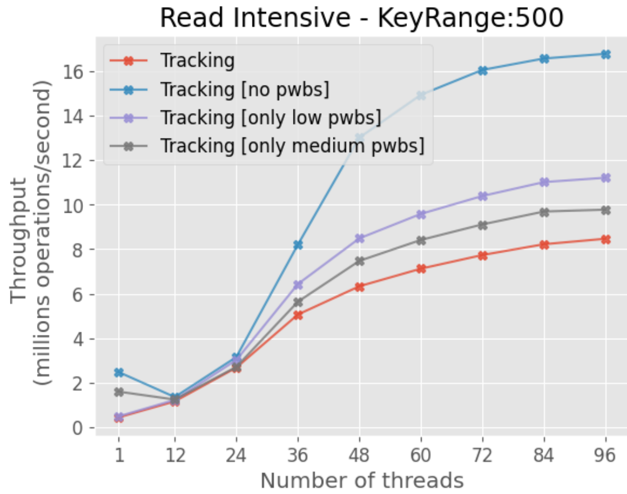
Thus, we have three sets of code lines, namely L, M and H, containing the code lines that have low, medium, and high performance impact, respectively. Figures 3e and 4e, show that Tracking mostly performs low-cost pwbs. Just a few of them are of medium cost, whereas no pwbs belong in the high-cost category. 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 *CAS* will wait for outstanding store operations to complete, which will cause the cache line affected by the pwb to be flushed and invalidated in cache. Then, this cache line is re-fetched for executing the actual update of the *CAS*, resulting in performance overhead. In case many threads issue first a pwb instruction and then a *CAS*, on the same cache line, the above scenario will occur repeatedly. Similar performance overheads may arise when pwbs are executed (by different threads) *after* a *CAS* (on the same cache line), as these pwbs will cause cache misses (and increased traffic on the memory controller). A psync (or pfence) instruction following pwbs 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 Tracking and 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 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 pwbs, 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 pwbs 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 pwbs of the other categories. Moreover, we observe that the removal of the pwbs of the category H has the biggest impact in the performance of Capsules-Opt. Thus, the diagrams show that Tracking owes its good performance to the fact that it executes just a few medium-cost pwbs. On the contrary, Capsules-Opt performs a lot of high-cost pwbs. The cost of the pwbs of category M is the dominant persistence cost in Tracking. Capsules-Opt performs less pwbs of this category and their combined impact is not high.

Summarizing, our results suggest that in a machine with Intel Optane, measuring only the number of pwbs is not enough to fully understand the persistence cost of a recoverable algorithm. A thorough evaluation is required since the impact of different pwbs 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



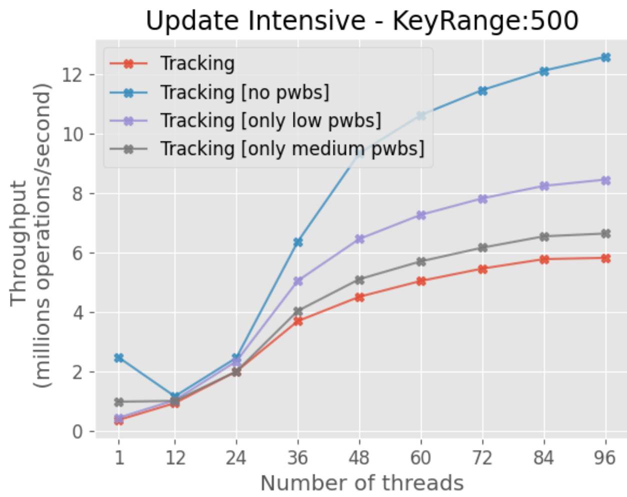
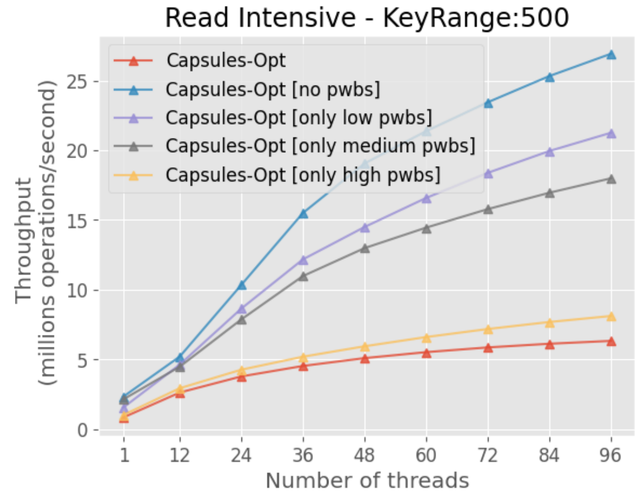


Figure 5. Impact of pwb categories on performance of Tracking.



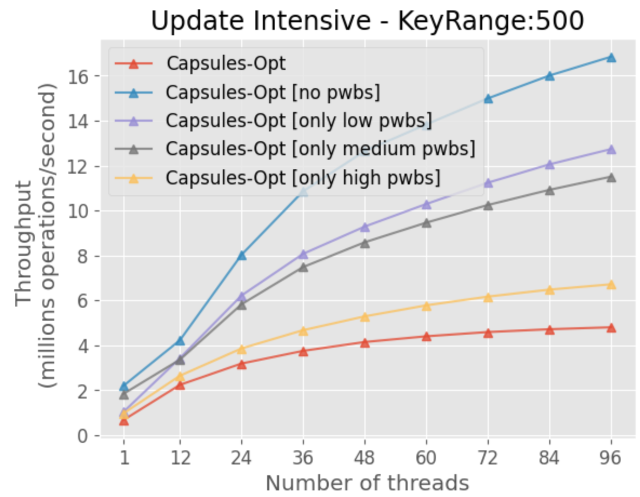


Figure 6. Impact of pwb categories on performance of Capsules-Opt.

improving the persistence cost of their algorithms. Additionally, experiments like those presented in Figures 3f and 4f, are more useful for performing an in depth comparison of the persistence cost of different algorithms.

Although in the algorithms we study, the 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 psync as well (recall e.g., that a psync instruction following pwbs 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.

Detectably Recoverable Binary Search Tree. The algorithm in [19] (LF-BST) implements a *leaf-oriented* (*external*) binary search tree. It uses *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 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

```
type Internal {                    ▹ subtype of Node
    Key ∪ {∞1, ∞2} key
    Node *left, *right
    Info *info
}
type Leaf {                        ▹ subtype of Node
    Key ∪ {∞1, ∞2} key
}
▹ Initialization:
Internal *Root := pointer to new Internal node with key field ∞2,
        info field NULL, and pointers to new Leaf nodes with keys ∞1
        and ∞2, respectively, as left and right fields.
        ▹ Assume the set contains two special values, ∞1 < ∞2, such that
        every other key k, ∞1 < k < ∞2
```

Figure 7. BST type definitions and initialization.

search tree implementation appear in Figure 7. Algorithms 5 and 6 present the code for Insert and Delete.

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 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 (code is provided in [2]). 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.

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, *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. The code is provided in [2].

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  RD_q := ⊥
4  pbarrier (RD_q)
5  CP_q := 1          // check-point; RD_q is initialized
6  pwb (CP_q); psync
7  while true do
8      Gather Phase // search for location to insert
9          ⟨–, p, l, –, pInfo⟩ := SEARCH(k)
10         AffectSet := {⟨p,pInfo⟩}
11     Helping Phase      // help other if necessary
12         if isTagged(pInfo) then
13             HELP (pInfo); continue
14     newSibling := a new Leaf whose key is l.key  // make
         a duplicate of l
15     newInternal := a new Internal node with key field
         max(k, l.key), and with two child fields equal to new
         and newSibling (the one with the smaller key is the
         left child)
16     if l = p → left then
17         WriteSet := {⟨p → left, l, newInternal⟩}
18     else WriteSet := {⟨p → right, l, newInternal⟩}
19     NewSet := {newInternal}
20     *opInfo := pointer to a new operation descriptor ⟨
         INSERT, AffectSet, WriteSet, NewSet, ⊥⟩
21     newInternal → info := getTagged(opInfo)
22     if l → key = k then      // key k is in the tree
23         opInfo → result := false
24     pbarrier (newSibling, newInternal, *opInfo)
25     RD_q := opInfo          // info for current attempt
26     pwb (RD_q); psync
27     if l → key = k then return false
28     HELP(opInfo, true)
29     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⟩
```

Algorithm 6: Recoverable BST: Delete

```
Procedure boolean DELETE (Key k)
40 Info *opInfo := new Info ()
41 RD_q := ⊥
42 pbarrier (RD_q)
43 CP_q := 1          // check-point; RD_q is initialized
44 pwb (CP_q); 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     RD_q := opInfo       // info for current attempt
64     pwb (RD_q); psync
65     if l → key ≠ k then return false
66     HELP(opInfo, true)
67     if opInfo → result ≠ ⊥ then return opInfo → result
```

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 [20, 21, 25, 34, 42, 44], heaps [21], hash maps (see e.g., [36, 52]), and B-trees (see e.g., [11, 37, 46]), 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], which we hope to investigate in future work.

Acknowledgments

We would like to thank all the anonymous reviewers for their helpful comments.

References

[1] Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. 2020. Tracking in Order to Recover - Detectable Recovery of Lock-Free Data Structures *(SPAA '20)*. Association for Computing Machinery, New York, NY, USA, 503–505. https://doi.org/10.1145/3350755.3400257

[2] Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. 2022. Tracking in Order to Recover: Detectable Recovery of Lock-Free Data Structures. arXiv:1905.13600 [cs.DC]

[3] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. 2018. Nesting-Safe Recoverable Linearizability: Modular Constructions for Non-Volatile Memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*. 7–16. https://doi.org/10.1145/3212734.3212753

[4] Greg Barnes. 1993. A Method for Implementing Lock-Free Shared-Data Structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '93, Velen, Germany, June 30 - July 2, 1993*. 261–270. https://doi.org/10.1145/165231.165265

[5] Ohad Ben-Baruch, Danny Hendler, and Matan Rusanovsky. 2020. Upper and Lower Bounds on the Space Complexity of Detectable Objects. In *Proceedings of the 39th Symposium on Principles of Distributed Computing* (Virtual Event, Italy) *(PODC '20)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/3382734.3405725

[6] Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. 2019. Delay-free concurrency on faulty persistent memory. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*. 253–264.

[7] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2016. Makalu: Fast Recoverable Allocation of Non-volatile Memory. *SIGPLAN Not.* 51, 10 (Oct. 2016), 677–694. https://doi.org/10.1145/3022671.2984019

[8] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A General Technique for Non-blocking Trees. In *Proc. 19th ACM Symposium on Principles and Practice of Parallel Programming*. 329–342.

[9] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. *PVLDB* 8, 5 (2015), 497–508. https://doi.org/10.14778/2735479.2735483

[10] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *PVLDB* 8, 7 (2015), 786–797. https://doi.org/10.14778/2752939.2752947

[11] Ping Chi, Wang-Chien Lee, and Yuan Xie. 2014. Making B^+-tree efficient in PCM-based main memory. In *International Symposium on Low Power Electronics and Design, ISLPED'14, La Jolla, CA, USA, August 11-13, 2014*. 69–74. https://doi.org/10.1145/2627369.2627630

[12] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*. 105–118. https://doi.org/10.1145/1950365.1950380

[13] Nachshon Cohen, Michal Friedman, and James R. Larus. 2017. Efficient logging in non-volatile memory by exploiting coherency protocols. *PACMPL* 1, OOPSLA (2017), 67:1–67:24. https://doi.org/10.1145/3133891

[14] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. 2018. The Inherent Cost of Remembering Consistently. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*. 259–269. https://doi.org/10.1145/3210377.3210400

[15] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*. 271–282. https://doi.org/10.1145/3210377.3210392

[16] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2020. Persistent memory and the rise of universal constructions. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo Seltzer (Eds.). ACM, 5:1–5:15. https://doi.org/10.1145/3342195.3387515

[17] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. 373–386. https://www.usenix.org/conference/atc18/presentation/david

[18] Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. 2014. The amortized complexity of non-blocking binary search trees. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*. 332–340. https://doi.org/10.1145/2611462.2611486

[19] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking binary search trees. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*. 131–140. https://doi.org/10.1145/1835698.1835736

[20] Panagiota Fatourou, Nikolaos D. Kallimanis, and Eleftherios Kosmas. 2021. Brief Announcement: Persistent Software Combining. In *35th International Symposium on Distributed Computing (DISC 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 209)*, Seth Gilbert (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 56:1–56:4. https://doi.org/10.4230/LIPIcs.DISC.2021.56

[21] Panagiota Fatourou, Nikolaos D. Kallimanis, and Eleftherios Kosmas. 2022. The Performance Power of Software Combining in Persistence. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, South Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, to appear.

[22] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. 2019. Persistent Non-Blocking Binary Search Trees Supporting Wait-Free Range Queries. In *Proc. 31st ACM on Symposium on Parallelism in Algorithms and Architectures*. 275–286.

[23] Steven Feldman, Carlos Valera-Leon, and Damian Dechev. 2016. An efficient wait-free vector. *IEEE Transactions on Parallel and Distributed Systems* 27, 3 (2016), 654–667.

[24] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 377–392. https://doi.org/10.1145/3385412.3386031

[25] Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. 2018. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*. 28–40. https://doi.org/10.1145/3178487.3178490

[26] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. Mirror: Making Lock-Free Data Structures Persistent. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1218–1232. https://doi.org/10.1145/3453483.3454105

[27] Wojciech M. Golab and Danny Hendler. 2017. Recoverable Mutual Exclusion in Sub-logarithmic Time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*. 211–220. https://doi.org/10.1145/3087801.3087819

[28] Wojciech M. Golab and Aditya Ramaraju. 2016. Recoverable Mutual Exclusion. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*. 65–74. https://doi.org/10.1145/2933057.2933087

[29] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings*. 300–314. https://doi.org/10.1007/3-540-45414-4_21

[30] Maurice Herlihy and Nir Shavit. 2008. *The art of multiprocessor programming*. Morgan Kaufmann.

[31] Intel. 2016. Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 3A. Available at https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html (September 2016).

[32] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*. 313–327. https://doi.org/10.1007/978-3-662-53426-7_23

[33] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Proceedings of the 30th International Symposium of Distributed Computing* (Vienna, Austria) *(DISC '16, Vol. LNCS 9888)*. Springer, 313–327.

[34] Prasad Jayanti and Anup Joshi. 2017. Recoverable FCFS Mutual Exclusion with Wait-Free Recovery. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*. 30:1–30:15. https://doi.org/10.4230/LIPIcs.DISC.2017.30

[35] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. 2018. IDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (Fukuoka, Japan) *(MICRO-51)*. IEEE Press, 258–270. https://doi.org/10.1109/MICRO.2018.00029

[36] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8 (apr 2020), 1147–1161. https://doi.org/10.14778/3389133.3389134

[37] Iulian Moraru, David G Andersen, Michael Kaminsky, Nathan Binkert, Niraj Tolia, Reinhard Munz, and Parthasarathy Ranganathan. 2012. Persistent, protected and cached: Building blocks for main memory data stores. *CMU Parallel Data Lab Trechnical Report, CMU-PDL-11-114 (Dec. 2011)* (2012).

[38] Aravind Natarajan, Arunmoezhi Ramachandran, and Neeraj Mittal. 2020. FEAST: a lightweight lock-free concurrent binary search tree. *ACM Transactions on Parallel Computing* 7, 2 (May 2020).

[39] Matej Pavlovic, Alex Kogan, Virendra J. Marathe, and Tim Harris. 2018. Brief Announcement: Persistent Multi-Word Compare-and-Swap. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*. 37–39. https://doi.org/10.1145/3212734.3212783

[40] PMDK. [n.d.]. The Persistent Memory Development Kit. https://github.com/pmem/pmdk/. https://github.com/pmem/pmdk/

[41] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. 2019. OneFile: A Wait-Free Persistent Transactional Memory. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*. IEEE, 151–163. https://doi.org/10.1109/DSN.2019.00028

[42] Matan Rusanovsky, Hagit Attiya, Ohad Ben-Baruch, Tom Gerby, Danny Hendler, and Pedro Ramalhete. 2021. Flat-Combining-Based Persistent Data Structures for Non-volatile Memory. In *Stabilization, Safety, and Security of Distributed Systems*, Colette Johnen, Elad Michael Schiller, and Stefan Schmid (Eds.). Springer International Publishing, Cham, 505–509.

[43] William N Scherer III, Doug Lea, and Michael L Scott. 2005. A scalable elimination-based exchange channel. *SCOOL 05* (2005), 83.

[44] Gal Sela and Erez Petrank. 2021. *Durable Queues: The Second Amendment*. Association for Computing Machinery, New York, NY, USA, 385–397. https://doi.org/10.1145/3409964.3461791

[45] Shahar Timnat and Erez Petrank. 2014. A practical wait-free simulation for lock-free data structures. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*. 357–368. https://doi.org/10.1145/2555243.2555261

[46] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*. 61–75. http://www.usenix.org/events/fast11/tech/techAbstracts.html#Venkataraman

[47] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[48] Ivan Walulya and Philippas Tsigas. 2017. Scalable Lock-Free Vector with Combining. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. 917–926. https://doi.org/10.1109/IPDPS.2017.73

[49] Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. 461–472. https://doi.org/10.1109/ICDE.2018.00049

[50] Yi Xu, Joseph Izraelevitz, and Steven Swanson. 2021. Clobber-NVM: Log Less, Re-Execute More. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 346–359. https://doi.org/10.1145/3445814.3446730

[51] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 465–477.

[52] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient Lock-Free Durable Sets. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 128 (oct 2019), 26 pages. https://doi.org/10.1145/3360554

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: $./\langle executable\rangle\ \langle algorithm\rangle\ [threads_number]\ [duration(seconds)]$

A.3 License

This code is provided under the LGPL-2.1 License.