Persistent Software Combining

Panagiota Fatourou
Université Paris Cité, LIPADE, France
FORTH ICS and University of Crete, Greece
faturu@csd.uoc.gr

Nikolaos D. Kallimanis
FORTH ICS
nkallima@ics.forth.gr

Eleftherios Kosmas
University of Crete, Greece
ekosmas@csd.uoc.gr

FORTH ICS TR 480 October 2022

Abstract

The availability of Non-Volatile Main Memory (known as NVMM) enables the design of recoverable concurrent algorithms. We study the power of software combining in achieving recoverable synchronization and designing persistent data structures. *Software combining* is a general synchronization approach, which attempts to simulate the ideal world when executing *synchronization requests* (i.e., requests that must be executed in mutual exclusion). A single thread, called the *combiner*, executes all active requests, while the rest of the threads are waiting for the combiner to notify them that their requests have been applied. *Software combining* significantly decreases the synchronization cost and outperforms many other synchronization techniques in various cases.

We identify three persistence principles, crucial for performance, that an algorithm's designer has to take into consideration when designing highly-efficient recoverable synchronization protocols or data structures. We illustrate how to make the appropriate design decisions in all stages of devising recoverable combining protocols to respect these principles. Specifically, we present two recoverable software combining protocols, satisfying different progress properties, that are many times faster and have much lower persistence cost than a large collection of existing persistent techniques for achieving scalable synchronization. We build fundamental recoverable data structures, such as stacks and queues, based on these protocols that outperform *by far* existing recoverable implementations of such data structures. We also provide the first recoverable implementation of a concurrent heap and present experiments to show that it has good performance when the size of the heap is not very large.

1 Introduction

Recent advances in memory technology have resulted in byte-addressable Non-Volatile Main Memory (NVMM), which attempts to combine the performance benefits of conventional main memory with the strong persistence characteristics of secondary storage. A program running in a traditional memory hierarchy system stores its operational data in volatile data structures maintained in DRAM, whereas its recovery data (such as transactional logs) are usually stored in non-volatile secondary storage. In the event of a failure, all in-memory data structures are lost and must be re-constructed from recovery data to make the system functional again. This poses major performance overheads. The availability of NVMM enables the design of concurrent algorithms, whose execution will be recoverable at no significant cost. An algorithm is *recoverable* (also known as *persistent* [14] or *durable* [52]) if its state can be restored after recovery from a system-crash failure. Another important property, known as *detectability* [6, 28, 38], is to be able to determine, upon recovery, if an operation has been completed, and if yes, to find its response. Despite many efforts for designing efficient recoverable synchronization protocols and data structures (see Section 7), persistence comes at a significant cost even for fundamental data structures, such as stacks and queues.

When designing recoverable algorithms, the main challenge stems from the fact that data stored into registers and caches are volatile. Thus, unless they have been flushed to persistent memory, such data will be lost at a system crash. Flushing to persistent memory occurs by including specific *persistence instructions*, such as `pwb`, `pfence` and `psync` in the code, which are however expensive in terms of performance.

In this paper, we reveal the power of software combining in achieving recoverable synchronization and designing persistent data structures. In *software combining* [22, 31, 42, 24, 37], each thread first announces its request, and then tries to become the combiner by acquiring a lock. The combiner applies several active requests, in addition to its own, before it releases the lock. As long as the combiner serves active requests, other threads perform local spinning, waiting for the combiner to release the lock. As soon as the lock is released, waiting threads whose requests have been served by the combiner, return the calculated responses, whereas the rest compete again for the lock. Software combining [22, 24] has been proved to outperform many other synchronization techniques in various cases, and has been used to implement state-of-the-art fundamental concurrent data structures, such as queues and stacks [22, 24], that lie in the heart of inter-thread communication mechanisms.

Although simple in their nature, combining protocols should be designed carefully, as they encompass five design decisions that all may have crucial impact in performance. Existing combining protocols differ in these design decisions, exhibiting different performance [22, 31, 37, 42].

Definition 1 *Design decisions for combining protocols that are crucial for performance:*

1. *the mechanism to decide which of the active threads will act as the combiner (e.g., some combining protocols use CAS [31, 21, 23], others use queue locks [22]);*
2. *the data structure to store the active requests;*
3. *how the updates are applied (e.g., directly on the shared state or on a copy of it);*
4. *the mechanism for collecting the requests' responses;*
5. *how to discover which requests have not been applied.*

In this paper, we present two *recoverable* software combining protocols, PBcomb which is blocking, and PWFcomb which is wait-free. We designed all five stages of our protocols taking into consideration three principles for reducing persistence cost (motivated by our experiments;

also discussed in [50, 54, 56, 4, 5]), that are presented in Definition 2. Our experiments show that the resulting protocols are many times faster than a large collection of existing persistent techniques for achieving scalable synchronization.

Definition 2 *Persistence principles crucial for performance:*

1. *The number of the persistence instructions should be maintained as low as possible. This encompasses that an implementation must store in NVMM only those variables (and persist those values of them) that are necessary for recoverability.*

2. *The persistence instructions should be of low cost. Not all persistence instructions have the same cost [5, 50, 56]. For instance, reducing contention on non-volatile variables can be beneficial for performance [5, 50].*

3. *Data to be persisted should be placed in consecutive memory addresses, so that they are persisted all together [54].*

Combining is a promising approach for achieving persistent synchronization at low cost, as having no more than the combiner thread persisting updates on the state of the implemented object is expected to reduce the number of persistence instructions that are performed, as well as to decrease contention on persisted data. However, the design decisions of state-of-the-art combining protocols [22, 31, 24, 37] are not fully in favor of supporting persistence in an efficient way: All these protocols store the active requests in a dynamic linked list, and have the combiner traversing the list to figure out which requests are active. Moreover, the combiner applies the active requests on the shared state of the object, and records responses in the list nodes. When attempting to make these protocols recoverable without changing their design decisions, the updated shared state, and the requests' responses that the combiners calculate need to be persisted for ensuring recoverability. These data are scattered in memory. This violates persistence principles 1 and 3, introduces several complications that the designer needs to cope with (see e.g., [47]), and results in high persistence overhead (see Section 6).

Our algorithms differ from existing state-of-the-art combining protocols (including the CC-Synch [22] algorithm and flat-combining [31]), illustrating how all five design decisions should take into consideration the three persistence principles of Definition 2. This results in protocols that have low persistence cost, in addition to being highly efficient in terms of synchronization. Our experiments show that both, PBcomb and PWFcomb, outperform by far, many previous recoverable Transactional Memory (TM) Systems [17, 18, 44, 45] and several generic mechanisms for designing recoverable data structures [4, 3, 10, 50] proposed in the literature. Specifically, PBcomb is 4x faster and PWFcomb is 2.4x faster than the competitors. Our protocols satisfy *detectable recoverability* [28], whereas most competitors (all but [10, 3]) guarantee only weaker consistency properties, such as *durable linearizability* [35].

We build recoverable queues and stacks using PBcomb and PWFcomb. Our experiments illustrate that the recoverable queues (PBqueue and PWFqueue) and stacks (PBstack and PWFstack) that are built on top of PBcomb and PWFcomb, have much better performance than state-of-the-art recoverable implementations of such data structures, including the specialized recoverable queue implementations in [28, 50]. Concurrent queues and stacks play a significant role in runtime systems [1], high performance computing [30, 2], kernel schedulers, network interfaces [43], etc. The proliferation of NVMM and the availability of highly-efficient recoverable stacks and queues could enable persistence in such settings.

Based on PBcomb, we were able to design the first recoverable concurrent heap (PBheap); experiments show that PBheap has good performance when the heap is not too large. PBheap is useful for implementing recoverable versions of algorithms that rely on priority queues when the problem input size is small or medium. Implementations of concurrent heaps often do not scale well due to contention (mainly at the root node). This makes a heap implementation a natural candidate for applying software combining.

Our contributions are summarized as follows.

- We present two highly-efficient recoverable combining protocols, which exhibit low persistence overhead and small synchronization cost.
- Experiments show that our protocols outperform *by far* state-of-the-art recoverable universal constructions and software transactional systems (that often ensure weaker consistency properties than our algorithms).
- We illustrate how to make the appropriate design decisions in all stages of designing combining protocols to respect the three persistence principles, crucial for performance. Our experiments reveal the performance power of respecting these principles.
- We built recoverable queues and stacks, based on our combining protocols, which outperform *by far* previous recoverable implementations of stacks and queues, including specialized recoverable implementations of such data structures [28, 47, 50].
- We provide the first recoverable implementation of a concurrent heap and present experiments to show that, for small/medium heap sizes, it has good performance.

2 Preliminaries

We consider a standard asynchronous distributed system with n threads. The system supports the atomic execution of *base primitives*, such as reads, writes, *CAS*, and *LL/VL/SC* on single-word shared variables. A $CAS(O, old, new)$ checks if the state of object O is equal to *old* and if so, it changes it to *new* and returns `true`, otherwise the state of O remains unchanged and `false` is returned. An *LL/SC* object O supports the operations *LL* (which returns the current value of O) and *SC*. By executing $SC(O, v)$, a thread p attempts to set the value of O to v. This change takes place only if no thread has changed the value of O (by executing *SC*) since the execution of p's latest *LL* on it; then, the *SC* is successful and returns `true`. Otherwise the *SC* returns `false`. We assume the Total Store Order (TSO) model, supported by x86 and SPARC, where writes by the same thread become visible in program order.

Current architectures supporting non-volatile main memory (e.g., those supporting Intel Optane DC Persistent Memory) provide both DRAM and NVMM. System-wide crash failures may occur at any point in time. When a failure occurs, the values of all variables stored in volatile memory (e.g., in registers, caches, or DRAM) are lost (upon recovery, these variables have their initial values), whereas values that have been written back (or *persisted*) to NVMM are non-volatile. Storing data in DRAM is desirable for good performance (Persistence Principle 1).

We assume *explicit epoch persistency* [35]: a write-back to persistent memory is triggered by a persistent write-back (`pwb`) instruction. The order of `pwb`s is not necessarily preserved. When ordering is required, a `pfence` instruction can be used to order preceding `pwb` instructions before all subsequent `pwb`s. A thread executing a `psync` instruction blocks until all previous `pwb` instructions complete. For each shared variable, `pwb`s preserve program order. We call `pwb`, `pfence`, and `psync`, the *persistence instructions*.

Failed threads can be recovered by the system in an asynchronous way. A *recoverable* (or *persistent*) implementation provides, for each thread and for each supported operation *op*, an associated *recovery function*. Upon recovery, *op*'s recovery function is invoked by the system for each thread that was executing an instance of *op* at the time the system crashed. If a crash occurs while the recovery function of *op* is executed, the recovery function of *op* is re-invoked.

An execution is *durably linearizable*, if the effects of all operations that have completed before a system crash are reflected in the object's state upon recovery (see [35] for a formal definition). *Detectability* [6, 28, 38] ensures that it is possible to determine, upon recovery, whether an

operation took effect, and its response value, if it did. *Detectable recoverability* ensures durable linearizability and detectability.

Detectable recoverability cannot be achieved without system support [9]. As in [9, 5, 47], we assume that the system persists the information that is needed for calling, for every thread p, the recovery function for p with the same arguments as the instance of *op* that p was executing at crash time. Moreover, for compatibility with previous work [28] (and fair treatment of the algorithms in the experimental analysis), we assume that each thread p has an associated persistent sequence number *seq* which it increments each time it invokes an operation *op* and passes it as a parameter to *op*. The system invokes the recovery function for *op* passing the same value for *seq* as in the original invocation of *op* by p. We remark that our algorithms also work with just passing to each operation of p a toggle bit (instead of *seq*) whose values alternate from one invocation of the thread to the next (i.e., just using the value of the last bit of *seq*). Our algorithms can be adjusted to work also with other assumptions for system support that have been made in previous work [4, 9] for designing detectable implementations (see Section 7 for more details). Without any system support, our algorithms ensure durable linearizability (but not 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. An execution is *wait-free*, if every operation completes within a finite number of steps if it does not experience any crash after some point of its execution.

3 Blocking Combining and Recoverability

Overview of PBcomb**.** PBcomb follows the general idea of blocking software combining [22, 31, 42, 24, 37]. PBcomb achieves low synchronization cost, while respecting all persistence principles:

1. PBcomb implements the lock in volatile memory. We have chosen a lock implementation which aims mainly at reducing synchronization cost. Moreover, the lock implementation allows a thread to leave the entry-section without ever acquiring the lock, if it finds out that its request has been served by a combiner.

2. PBcomb utilizes an array, *Request*, to store the threads' requests in consecutive memory addresses. This array is stored in volatile memory (i.e., it does not have to be persisted). This results in lower persistence cost.

3. Each combiner creates a copy of the state of the implemented object and applies the active requests on this copy (and not on the shared state of the object). This is one of the most crucial design decisions of PBcomb in terms of performance. The combiner switches a shared variable to index the copy it used, indicating that it stores the current valid state of the implemented object. The combiner should persist the copy it used before trying to switch the pointer.

   There is an interesting performance tradeoff between the approach of performing updates directly on the shared state and that of creating a copy of the state to apply the updates on. In the first technique, the updates are performed on data that are usually scattered in memory. Persisting the updated values is thus expensive. This problem is avoided by the second technique which persists data stored in the copy in consecutive memory addresses. However, the second technique works well mainly for objects of small or medium size (or when the number of synchronization points is small). In other cases, the cost of copying and persisting the state may dominate the cost of persisting a smaller amount of scattered data (part of the state).

A well-known limitation [21, 22, 24] of the combining technique is that using a single thread to apply all active requests may restrict parallelism, if the size of the object or the number of synchronization points are large. PBcomb (similarly to previous persistent algorithms [47] that are based on some combining protocol), inherits the limitations of the technique. Thus, PBcomb works well mainly for implementing objects of small and medium size or when the number of synchronization points is small (as is the case with stacks and queues). Subsequently, creating a copy of the state significantly reduces the persistence cost without imposing any additional limitation to the algorithm.

4. Following persistence principle 3, PBcomb stores the response values in an array, maintained together with the state of the object (in consecutive memory addresses). The combiner persists the entire array of return values together with the object's state.

5. PBcomb uses two bits (*activate* and *deactivate*) for each thread p, to identify whether the last request, initiated by p, has been served. If the two bits are not equal, p has a request which has not yet been served i.e., it is *active*. PBcomb persists just the deactivate bit of p. Following persistence principle 3, PBcomb stores the deactivate bits together with the object's state, so all data to be persisted are in consecutive memory locations.

We define the *combining degree*, d, to be the average number of requests that a combiner serves. PBcomb executes a small number of `pwb` instructions for every d requests. Moreover, in PBcomb, threads other than the combiner do not have to execute any persistence instructions. Additionally, the combiner does not persist each of the requests it applies separately; data to be persisted are stored in consecutive memory addresses and are persisted all together. Thus, PBcomb respects the persistence principles, maintaining persistence cost low. Additionally, PBcomb has significantly lower synchronization cost than previous combining protocols [21, 22, 31], as well as than its competitors; this is another major reason for its good performance.

Detailed Description. PBcomb appears in Algorithms 1 and 2. Each element of *Request* stores a *RequestRec* record with fields: i) a pointer *func* to a function to execute in order to serve the request, ii) a set *args* of arguments to *func*, iii) a bit *activate* used to identify whether the request has already been served or not, and iv) a *valid* bit used for ensuring recoverability. A request that has not experienced a crash, has its *valid* bit equal to 1, whereas at recovery time, this bit is reinitialized to the value 0. At recovery time, this bit is used to disallow a combiner to re-execute a request that has already been executed before the crash. A request whose *valid* bit is equal to 1 is called *valid*.

PBcomb maintains two records of type *StateRec* in array *MemState*. It uses them to store copies of the object's state. The current state of the implemented object is stored in the element of *MemState* indexed by the variable *MIndex*. Each record of type *StateRec* comprises a field *st* storing the object's state, and two arrays. The first, *ReturnVal*, stores, for each thread, a response for the last request initiated by the thread. The second is the *deactivate* bit vector.

To reduce the synchronization cost, the implementation of the lock in PBcomb is different than in existing combining protocols [22, 24, 31, 37]. PBcomb uses an integer shared variable *Lock*: an odd value stored in it indicates that the lock is taken, whereas an even value indicates that the lock is free. Implementing the lock in this way, allows a thread q to wait on line 10, each time it executes it, only for the thread p that was the current combiner the last time q accessed *Lock*. Moreover, q can leave the entry-section without executing *CAS*, if it discovers that its request has been served. Additionally, for each set of combined operations, a single successful *CAS* is executed. Lock implementations in which every thread should wait its turn to enter the critical-section before it leaves the entry-section (e.g., that in [48]) may negatively impact performance.

A thread p starts by recording its request, and (the reversed value of) its activate bit, in *Request*$[p]$. Next, it checks if the lock is acquired and if not, it tries to acquire it by executing a

Algorithm 1: PBcomb – Code for thread $p \in \{0, \ldots, n-1\}$

```
type RequestRec {
    Function func
    Argument Args
    Bit activate
    Bit valid
}
type StateRec {
    State st
    ReturnValue ReturnVal[0..n − 1]
    Bit Deactivate[0..n − 1]
}
▷ Shared non-volatile variables:
StateRec MemState[0..1], initially ⟨⊥, ⟨⊥, ..., ⊥⟩, ⟨0, ..., 0⟩⟩
Bit MIndex, initially 0
▷ Shared volatile variable:
RequestRec Request[0..n − 1], initially ⟨⟨⊥, ⊥, 0, 0⟩, ..., ⟨⊥, ⊥, 0, 0⟩⟩
Integer Lock, initially 0
Integer LockVal, initially 0

Procedure ReturnValue PBcomb(Function func, Argument Args, Integer seq)
    // Announce request
1   Request[p] := ⟨func, Args, 1 − Request[p].activate, 1⟩
2   return PerformRequest()

Procedure ReturnValue Recover(Function func, Argument Args, Integer seq)
3   Request[p] := ⟨func, args, seq mod 2, 1⟩
    // if request is not yet applied
4   if MemState[MIndex].Deactivate[p] ≠ seq mod 2 then
5       return PerformRequest()
6   return MemState[MIndex].ReturnVal[p]   // request is applied
```

CAS. If it succeeds, p becomes the combiner and starts executing the *combiner code* (lines 14-28). Every thread q that does not become the combiner busy waits until the current combiner has released the lock. Then, q checks whether its request has been served; this is true when q's activate and deactivate bits are equal. If so, q returns its response value. The remaining threads contend again for the lock.

In the combining code, a combiner p chooses among the two *StateRec* records of *MemState* the one, r, that is not indexed by *MIndex*, to use for serving requests. Then, in line 15, it copies the current state of the object into r. Next, it executes a for loop (*simulation phase*), where for each thread q: If there is an active, valid request by q, p a) applies the request using r, ii) records the response into the appropriate element of the *ReturnVal* array stored in r, and iii) changes *Deactivate*$[q]$ in r to make it equal to its activate bit. As soon as p completes the simulation phase, it changes *MIndex* to index r and unlocks *Lock* giving up its combining role.

We say that *req* has *taken effect* at some point t, if a combiner p a) has read *req* in *Request* by t, b) has performed line 18 for q *applying req*, and c) has executed line 27 by t. We discuss the correctness of PBcomb in [25]. To comply with persistence principle 1, PBcomb stores a number of its variables in DRAM. This results in improved performance. The algorithm can be easily modified to work correctly even if all data are stored in non-volatile memory.

Persistence. When Recover(*func*, *args*, *seq*) is called for a thread p, p first executes line 3, where it recovers its own entry in *Request*. This is necessary to appropriately set p's *activate* and *valid* bits. This way a combiner is disallowed to re-execute (or to avoid execute) p's request by seeing an erroneous initial value in p's *activate* bit after a crash. Next, p checks whether the last bit of *seq* is the same as *MemState*$[$*MemIndex*$]$.*deactivate*$[p]$. If yes, then the request has been executed and its response is returned. Otherwise, p re-invokes PBcomb(*func*, *args*, *seq*). Recall that we assume that the system calls Recover, for each recovered operation, with the same parameters as PBcomb.

Algorithm 2: PBcomb – Code for thread $p \in \{0, \ldots, n-1\}$

```
1  Procedure ReturnValue PerformRequest()
3      Bit ind                                                          // local variable of p
4      Integer lval                                                     // local variable of p
5      while true do
6          lval := Lock
7          if lval mod 2 = 0 then
8              if CAS(Lock, lval, lval + 1) = true then break
9              lval := lval + 1
10         wait until Lock ≠ lval
11         if Request[p].activate = MemState[MIndex].Deactivate[p] then
12             if LockVal ≠ lval then wait until Lock ≠ lval + 2
13             return Memstate[MIndex].ReturnVal[p];
14     ind := 1 − MIndex
15     MemState[ind] := MemState[MIndex]                                 // copy current state
16     for q ← 0 to n − 1 do
           // if q has a request that is not yet applied
17         if Request[q].activate ≠ MemState[ind].Deactivate[q] and Request[q].valid = 1 then
18             apply Request[q].func with Request[q].Args on MemState[ind].st
19             compute return value, returnVal
20             MemState[ind].ReturnVal[q] := returnVal
21             MemState[ind].Deactivate[q] := Request[q].activate
22     pwb(&MemState[ind])
23     pfence()
24     LockVal := Lock
25     MIndex := ind
26     pwb(&MIndex)
27     psync()
28     Lock := Lock + 1
29     return MemState[MIndex].ReturnVal[p]
```

We next explain the role of each of the persistence instructions of PBcomb. If the `pwb` instructions of lines 22 and 26 do not exist, a thread will have no way, at recovery time, to find the current state of the object, or its response. Additionally, a `pfence` (line 23) must exist between these `pwb`s: Assume that a crash occurs just after *MIndex* has been persisted (line 26). If no `pfence` exists between the `pwb`s of lines 22 and 26, then the `pwb` on *MemState*[*MIndex*] could be delayed and thus, the contents of *MemState*[*MIndex*].*st* may be partially persisted, at the time of the crash. (Note that *MemState*[*MIndex*] may be stored in more than one cache line.) Consider a request *req* by a thread q that has been served using *MemState*[*MIndex*].*st* and assume that the part of the state reflecting *req*'s updates, has not been persisted before the crash. Assume that the *Deactivate*[q] bit of *MemState*[*MIndex*] has been persisted before the crash. Then, at recovery time, the value of the last bit of *seq* is the same as *MemState*[*MIndex*].*Deactivate*[p], and *req* responds (line 6), thus violating durable linearizability.

Assume now that the `psync` of line 27 is missing. Consider an execution where a request *req* by a thread q has been applied by a combiner p. Assume that after p releases the lock, q responds for *req* (line 13). Then, if a crash occurs, it may happen that *MIndex* has not yet been persisted before this crash. At recovery, the object returns in some earlier state, thus violating durable linearizability.

The persistence of *ReturnVal* and *deactivate*, and the use of *seq*, are required only to ensure detectability. Consider a request *req* (initiated by thread q) that has taken effect, and assume that the system crashes before *req* completes, i.e., q should be able to find the response of *req* at recovery time. If *ReturnVal*[q] was not persisted, q could not find *req*'s response at recovery time, violating detectability.

The persistence of *deactivate*, as well as the use of *seq*, allow q to determine whether *req* took effect before a crash. In line 17, PBcomb compares q's activate and deactivate bits to determine whether *req* is still active. Note that persisting the activate bits (in addition to *deactivate*) would not be enough to determine whether *req* was still active when a crash occurred. Assume that

Algorithm 3: PWFcomb - Code for thread $p \in \{0, \ldots, n-1\}$

```
type RequestRec {
    Function func
    Argument args
    Bit activate
    Bit valid
}
type StateRec {
    State st
    ReturnValue ReturnVal[0..n − 1]
    Bit Deactivate[0..n − 1]
    Bit Index[0..n − 1]
    {0, 1, .., n − 1} pid
}
▷ Shared non-volatile variables:
StateRec MemState[0..n][0..1], initially ⟨⊥, ⟨⊥, . . . , ⊥⟩, ⟨0, . . . , 0⟩, ⟨0, . . . , 0⟩, 0⟩
StateRec *S := &MemState[n][0]
▷ Shared volatile variables:
RequestRec Request[0..n − 1], initially ⟨⟨⊥, ⊥, 0, 0⟩, . . . , ⟨⊥, ⊥, 0, 0⟩⟩
Integer Flush[0..n − 1], initially ⟨0, . . . , 0⟩
Integer CombRound[0..n − 1][0..n − 1], initially ⟨⟨0, . . . , 0⟩, . . . , ⟨0, . . . , 0⟩⟩

Procedure ReturnValue PWFcomb(Function func, Argument args,
Integer seq)
    // Announce request
1   Request[p] := ⟨func, args, 1 − Request[p].activate, 1⟩
2   Backoff()
3   return PerformRequest()

Procedure ReturnValue Recover(Function func, Argument args,
Integer seq)
4   Request[p] := ⟨func, args, seq mod 2, 1⟩
    // if request is not yet applied
5   if S → Deactivate[p] ≠ seq mod 2 then
6       return PerformRequest()
    // request is applied
7   return S → ReturnVal[p].ret
```

a thread q executes two consecutive requests, req_1 and req_2 of the same type with the same arguments, and the system crashes just before req_1 returns. Thread p cannot distinguish this situation from the case where req_2 has just been invoked, as these two bits will be equal in both cases. However, in the first case, q should return the response of req_1, whereas in the second, it should re-execute req_2. To be able to distinguish these two cases, additional (system) support is required [9]. Following previous work [28], PBcomb makes use of the *seq* parameter. To reduce persistence cost, PBcomb avoids persisting both *seq* and $Activate[q]$. As *seq* is provided by the system, there is no need for PBcomb to persist it; we let its last bit play the role of the activate bit at recovery, so PBcomb does not persist *activate*.

Note that in a durably linearizable version of PBcomb, the only field of *StateRec* that needs to be persisted is *st*. This will reduce the number of cache lines that need to be persisted (by the `pwb` of line 22). Moreover, the durable linearizable version of PBcomb has *null recovery* [35], i.e., no recovery function is necessary. Detectable recoverability for PBcomb is further discussed in [25].)

4 Wait-free Recoverable Combining

In PWFcomb (Algorithms 3 and 4), all threads pretend to be the combiner: they copy the state of the object locally and use this local copy to apply all active requests they see announced. Then, each of them attempts to change a pointer, S, to point to its own local copy using SC. If a thread p manages to do so, then p is indeed the thread that acted as the combiner. PWFcomb

Algorithm 4: PWFcomb - Code for thread $p \in \{0, \ldots, n-1\}$

```
1  Procedure ReturnValue PerformRequest()
3      StateRec *lsPtr
5      Integer lval
7      for l ← 1 to 2 do
9          lsPtr := LL(S)
11         Bit ind := lsPtr → Index[p]
13         MemState[p][ind] := *lsPtr                          // copy current state
14         MemState[p][ind].pid := p
15         lval := Flush[lsPtr → pid]
16         if lval mod 2 = 0 then lval := lval + 1
17         else lval := lval + 2
18         if VL(S) = false then continue
19         for q ← 0 to n − 1 do
               // if q has a request that is not yet applied
20             if Request[q].activate ≠ MemState[p][ind].Deactivate[q] and Request[q].valid = 1 then
21                 apply Request[q].func with Request[q].args on MemState[p][ind].st
22                 compute return value, returnVal
23                 MemState[p][ind].ReturnVal[q] := returnVal
24                 MemState[p][ind].Deactivate[q] := Request[q].activate
25                 CombRound[p][q] := lval
26         if VL(S) = true then
27             MemState[p][ind].Index[p] := 1 − MemState[p][ind].Index[p]
28             pwb(&MemState[p][ind])
29             pfence()
30             Flush[p] := lval
               // Try to change S content
31             if SC(S, &MemState[p][ind]) = true then
32                 pwb(&S)
33                 psync()
34                 CAS(&Flush[p], lval, lval + 1)
35                 return S → ReturnVal[p]
36             BackoffCalculate();
38     lsPtr := S
40     lval := Flush[lsPtr → pid]
42     if lval mod 2 = 1 and lval = CombRound[lsptr → pid][p] then
44         pwb(&S)
46         psync()
48         CAS(Flush[p], lval, lval + 1)
50     return S → ReturnVal[p]
```

borrows ideas from the universal constructions in [21, 23, 33], and can serve as a highly efficient, persistent version of these algorithms.

Similarly to PBcomb, PWFcomb uses a *Request* array and a *StateRec* record that contains the state of the implemented object, the array of *deactivate* bits and the array *ReturnVal*. PWFcomb maintains 2 records of type *StateRec* for each thread (in addition to two dummy such records, needed for correct initialization). This is necessary as each thread pretends to be the combiner: each thread has to use two *StateRec* records of its own, to copy the state of the object locally. Because of this, achieving recoverability is more complicated in PWFcomb than in PBcomb. The array *Index* of a *StateRec* record aims at coping with some of these complications. To ensure that persistence principles 1 and 2 (Definition 2) are respected, we use the *flush* integer and the *CombRound* array (more details below).

To execute a request *req*, a thread p announces *req* and calls PerformRequest to serve active requests (including its own). In PerformRequest, p reads S (line 9) and decides which of the two *StateRec* records in its pool it will use (line 11); this information is recorded in $Index[p]$ of the *StateRec* record pointed to by the value of S that p read. Next, it makes a local copy of the *StateRec* record pointed to by S (line 13). Making a local copy of the state is not atomic, thus p validates on line 18 that its local copy is consistent. Then, p proceeds to the simulation phase (which is similar to that of PBcomb). Afterwards, it executes an *SC* in an effort to update S to point to the *StateRec* on which it was working (line 31). Since PWFcomb

builds upon and extends PSim [21, 23] (see Section 7), proving its correctness (in the absence of failures) follows similar arguments as for PSim [23].

The recovery function of PWFcomb is the same as that of PBcomb. We focus on the persistence challenges that arise due to the recycling of *StateRec* records. The fact that a thread has two *StateRec* records and uses them alternatively, ensures that no thread ever performs active requests on the *StateRec* record pointed to by S. Thus, threads that read the currently active state see consistent data. A variable for each thread p, points to the *StateRec* record that p will use next. We store these variables into the *Index* array of *StateRec*, so that p persists them together with the *StateRec* it uses (at lines 28-29), in accordance to persistence principle 3. Persisting *Index* is necessary, since otherwise the following bad scenario may happen. Assume that one of the SC instructions executed by p is successful and let *ind* be the value that p reads on line 11, before the execution of SC. Assume also that the system crashes after p persists the new value of S and completes. Upon recovery, p discovers that its last request has been completed and invokes a new request. Then, it may happen that p chooses again the same record $MemState[p][ind]$, and start serving new requests on the current state of the object. Other active threads may, thus, read (on line 13) inconsistent data.

Before a thread p, that has initiated a request *req*, responds, p must persist the value of S. This should be done independently of whether p has successfully executed the SC of line 31, for the following reason. Assume that p responds for *req* without persisting S and then the system crashes. Upon recovery, S will point to a *StateRec* corresponding to some previous state of the object than that in which p read *req*'s response. This could violate detectable recoverability. For the same reason, executing just the `pwb` of line 32 (or 44) is not enough and the `psync` of line 33 (or 46) is also needed.

Experiments showed that having all threads performing a `pwb` and a `psync` to persist the contents of S before completing, results in high persistence cost. This is not surprising as this approach violates persistence principles 1 and 2. To respect these principles, arrays *Flush* and *CompRound* are used. *Flush* has one entry for each thread and it is used to indicate whether S has already been persisted or not, as described below. Consider a combiner p that successfully updates S (executing the SC of line 31). Before executing the corresponding SC, p changes $Flush[p]$ to an odd value (lines 15-17 and 30). Then, after (updating and) persisting S, p updates $Flush[p]$ to an even value (line 34), indicating that this change of S has already been persisted. All other threads persist S only if $Flush[p]$ contains an odd value (first condition of line 42), in which case, they update $Flush[p]$ to the next even value (line 48). The use of array *CombRound* allows a thread q to persist only the change of S performed by the combiner p that served q's request, as follows. For each thread whose request it serves, p stores in its row of *CombRound* the odd value that its $Flush[p]$ integer has when it executes the corresponding SC. Thread q persists S only if $Flush[p]$ contains $CombRound[p][q]$ (second condition of line 42). These techniques contribute to preserving persistence principles 1 and 2. To maintain persistence principle 1, we store both *Flush* and *CombRound* in volatile memory.

5 Recoverable Data Stuctures

Here, we provide summaries of our recoverable data structures. More details and pseudocodes are provided in [25].

PBStack and PWFstack. The stack is implemented as a linked list of nodes. Since the stack has a single point of synchronization, the state of the stack maintained by our algorithms is just the value of *top*, the pointer pointing to the topmost element of the stack. A combiner p copies the appropriate element of *MemState*, reads the current value of *top* from it, and serves the active requests using the element, r, of *MemState* that p has chosen to work on. To serve a Push, p has to additionally allocate a new node and set the next pointer of it to point to the value of *top* it read. The combiner persists the fields of all newly allocated nodes before persisting e (see also memory management below). The combiner applies elimination [32] to

pair off concurrent PUSH and POP operations without accessing the state of the object. This has small positive impact in performance (Figure 7a).

PBQueue. PBQUEUE uses a singly-linked list to store the nodes of the queue. To increase parallelism and enhance performance, we do not employ PBCOMB in an automatic way. We rather utilize two instances of PBCOMB, one to synchronize the enqueuers (I_E), and another to synchronize the dequeuers (I_D); thus, combiners of I_E serve only enqueue requests, while combiners of I_D serve only dequeue requests. This results in increased parallelism: enqueues are executed concurrently with dequeues (but sequentially to other enqueues). I_E stores just the queue's tail pointer in the *st* field of its *StateRec* records, whereas I_D stores just the head pointer. ENQUEUE and DEQUEUE operations add and remove nodes directly to and from the linked list that implements the queue. The first list node always plays the role of a dummy node.

The persistence scheme of PBCOMB guarantees that the head and the tail of the queue are persisted. PBQUEUE also persists the modifications performed by the combiners on the nodes of the linked list. This is necessary, since otherwise, these modifications will not survive after a crash, which may result in an inconsistent state and violate durable linearizability. A DEQUEUE only updates the head of the simulated queue and does not modify the nodes of the linked list. Therefore, the effects of a dequeue combiner on the simulated state of the queue are correctly persisted by the dequeue instance of PBCOMB. However, there is a subtlety that needs to be addressed regarding the nodes of the linked list that can be removed by the dequeue combiners. An enqueue combiner simulates the active ENQUEUE requests by directly modifying the nodes of the linked list and then persisting these modifications. Thus, if no care is taken, a dequeue combiner may remove list nodes that have been appended by an active enqueue combiner but not yet persisted. This may jeopardize detectable recoverability. To address this, PBQUEUE disallows dequeue combiners to remove any node from the linked list that has not yet been persisted. It achieves this by using a shared volatile variable *oldTail*. An enqueue combiner updates *oldTail* to point to the last node of the queue after it persists its changes and before releasing the lock. A dequeue combiner removes nodes from the linked list up until *oldTail*.

PWFQueue. PWFQUEUE combines ideas from PBQUEUE and SIMQUEUE [21, 23]. As in PBQUEUE, the queue is implemented as a singly-linked list and PWFQUEUE uses two instances of PWFCOMB (I_E and I_D) to synchronize the enqueuers and the dequeuers. A thread executing an ENQUEUE will also try to serve ENQUEUEs by other enqueuers. It does so by creating a local list of new nodes that will eventually be appended to the current state of the queue. So, at some point in time, the linked list implementing the queue may be comprised of two parts. To ensure consistency, all threads perform the linking of these parts before they proceed to serve requests. Also, the state maintained by I_E is now comprised of three pointers to support the linking of the two parts of the list.

Regarding persistence, some subtleties arise from the necessity to connect the two parts of the linked list representing the queue. Before updating the queue's tail, an enqueuer p has to persist the pointers needed to connect the two parts of the linked list, i.e., the current tail of the queue and the pointer to the first node of its local list. If it does not do so, then the system may crash just after p updates the tail (and before it connects the two parts of the linked list), in which case its local copy is lost and durable linearizability may be jeopardized. Additionally, an enqueuer that connects the linked list, has to persist the new values of the node it updated (i.e., its pointer to the next element of the linked list). Although dequeuers also help connecting the two parts of the list, it is enough to persist only the head of the queue.

The code and a more detailed description of PBQUEUE and PWFQUEUE are provided in [25].

PBHeap. PBHEAP is a persistent bounded min-heap implementation based on PBCOMB. The state stored in *StateRec* is the array of heap elements and two integers identifying the bounds of the heap. PBHEAP supports the operations HGETMIN, HINSERT, and HDELETEMIN. It employs a single instance of PBCOMB and is implemented by enhancing a sequential heap

implementation with the code of PBcomb.

Memory Management. For ensuring persistence in allocating new stack or queue nodes, we follow a standard technique [45, 17, 47] where each thread p pre-allocates a fixed-size memory chunk in NVMM, and reserves nodes from this chunk. Whenever this chunk is exhausted a new memory chunk is allocated by p. If a garbage collection mechanism (for collecting nodes) is not used, whenever p serves as the combiner, it gets nodes in consecutive memory addresses (to comply with the persistence principles).

For garbage collection, in PBqueue, each thread p has its own free list and places there nodes it removes when acting as combiner. It does so, after causing the removal of these nodes to take effect. Whenever p needs to reserve nodes while its free list is not empty, it uses nodes from this list. Note that this does not ensure persistence principle 3, as the nodes in its free list may belong to chunks of other threads. We were able to implement an efficient garbage collection scheme for PBstack (by exploiting its semantics). We maintain a single free list for all threads, implemented as a stack (*recycling stack*). Whenever p needs to reserve a node, it pops a node from the recycling stack. This way recycled nodes are re-inserted in the implemented data structure in the same order as they have originally been reserved from the memory chunk. This complies with persistence principle 3.

To support garbage collection in PWFstack, we extend the scheme described for PBqueue with the simple validation scheme of [11], which disallows a thread to access nodes that have already been placed in a free list. For PWFqueue, a solution would be more complicated, due to the fact that there may be two parts that comprise the state of the queue. We have left this for future work.

6 Performance Evaluation

We evaluate our algorithms on a 48-core machine (96 logical cores) consisting of 2 Intel Xeon Platinum 8260M processors with 24 cores each. Each core executes two threads concurrently. 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* [44], which provides the `pwb` and `psync` persistency instructions. An $x86\_64$ store fence instruction is used for implementing a `pfence` operation. The operating system is Linux (kernel v3.4) and we use *gcc* v9.1.0. Threads were bound in all experiments following a scheduling policy which distributes the running threads evenly across the machine's NUMA nodes [22, 23]. For our experiments, we simulate an LL on an object O with a read, and an SC with a CAS on a timestamped version of O to avoid the ABA problem. We executed each experiment 10 times (runs) and display averages. Each run simulates 10^7 atomic operations in total, with each of the n threads simulating $10^7/n$ operations. In the experiments for the stacks (queues), each thread performs pairs of Push and Pop (Enqueue and Dequeue) starting from an empty data structure. This experiment is kind of standard [21, 22, 23, 28, 47, 50], as it avoids performing unsuccessful (and thus cheap) operations. We performed also experiments where each thread executed random operations (50% of each type), as well as experiments where the data structure was initially populated; as they did not illustrate significant differences in the performance trends of the tested algorithms, we do not report these experiments.

Synthetic Benchmark. We first consider a synthetic benchmark (`AtomicFloat`) in which every thread, repeatedly, executes `AtomicFloat`(O, k) that reads the value v of O and updates it to $v * k$; the thread returns the value read. To avoid long runs and unrealistically low number of cache misses [22, 21, 24, 40], we added a local workload between consecutive executions of atomic operations, implemented as a short loop of a random number (maximum 512) of dummy iterations [22, 21]. In Figure 1, we compare the performance of `AtomicFloat` implementations based on PBcomb and PWFcomb against state-of-the-art wait-free persistent synchronization techniques: OneFile [45], CX-PUC [18], CX-PTM [18], and RedoOpt [18], using the latest version of code for these algorithms provided in [16]. These algorithms satisfy durable lineariz-

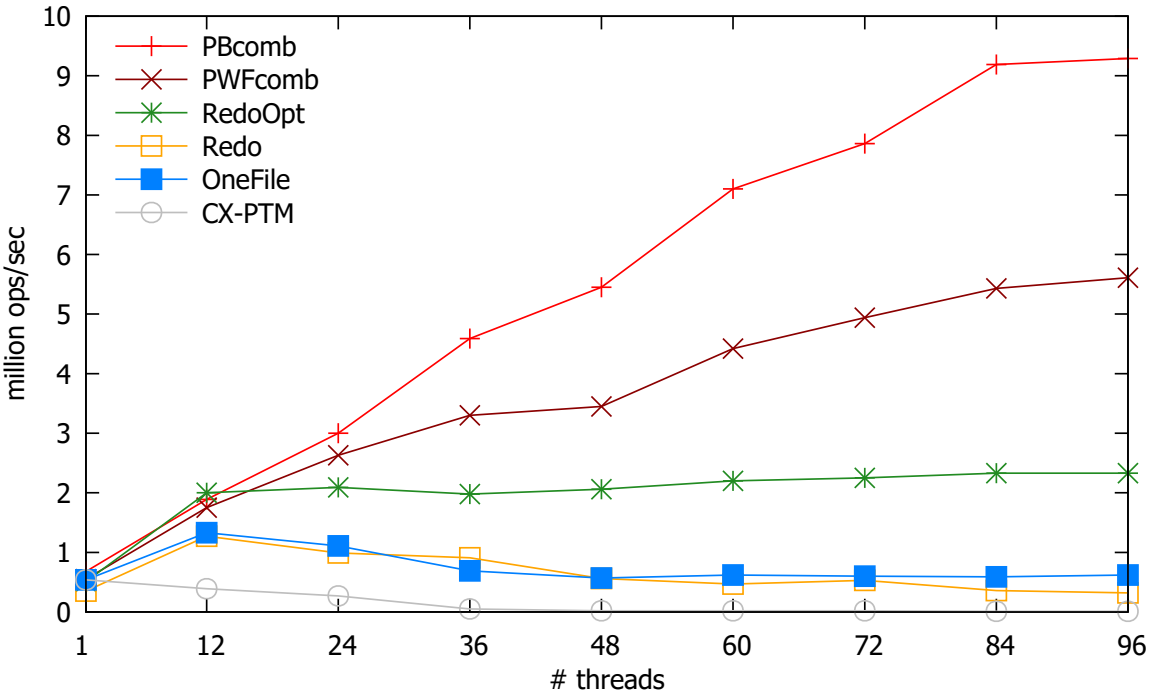


Figure 1: Simulation of a persistent `AtomicFloat` object on Intel Xeon: throughput.

ability (not detectable recoverability). Figure 1 shows that PBcomb is more than 4x faster than RedoOpt, which is the fastest among the competitors. Also, PWFcomb is more than 2.8x faster than RedoOpt. Figure 2 shows that both PBcomb and PWFcomb perform (on average) a small number of `pwb` instructions per operation. Figure 3 shows that the impact of `psync` is negligible in our experiments. This experiment illustrates that the main persistence cost in our algorithms comes from the `pwb` instructions, and reveals the importance of keeping the number of `pwb`s (and their cost) low, thus respecting persistence principles 1 and 2, when designing persistent synchronization protocols and concurrent data structures.

Note that PBcomb causes almost the same number of `pwb`s as RedoOpt [18]. RedoOpt uses ideas from PSim [21], and thus it employs some form of combining. Because of this, RedoOpt executes a low number of low cost `pwb` instructions. However, RedoOpt employs a shared queue, stored in volatile memory, to impose an order to the executed operations, which results in high synchronization overhead. PBcomb achieves the same number of `pfence`s and `psync`s as RedoOpt and does not cause any noteworthy increase to the number of `pwb`s. Interestingly, this is achieved at a much lower synchronization cost (see Figure 6, discussed below).

PBcomb performs better than PWFcomb in all experiments. The main reasons are that 1) the synchronization cost of PBcomb is lower than PWFcomb (see Figure 8), and 2) PWFcomb has higher persistence cost, as all threads should ensure that S is persisted before returning. These costs are paid to ensure the wait-free property of PWFcomb.

Persistent queues. Figure 4 compares the performance of PBqueue and PWFqueue with persistent queue implementations based on the persistence techniques studied in Figure 1. It also compares PBqueue and PWFqueue with the specialized persistent queue implementation in [28] (FHMP), and those recently published in [50] (OptLinkedQ and OptUnlinkedQ), as well as the persistent queue implementations based on Capsules-Normal [10] (NormOpt), and persistent queue implementations based on Romulus [17] (i.e., RomulusLR and RomulusLog). Figure 4 shows that PBqueue achieve superior performance by being 2x faster than the OptUnlinkedQ, which is the best competitor. Figure 5 shows the number of `pwb`s in differ-

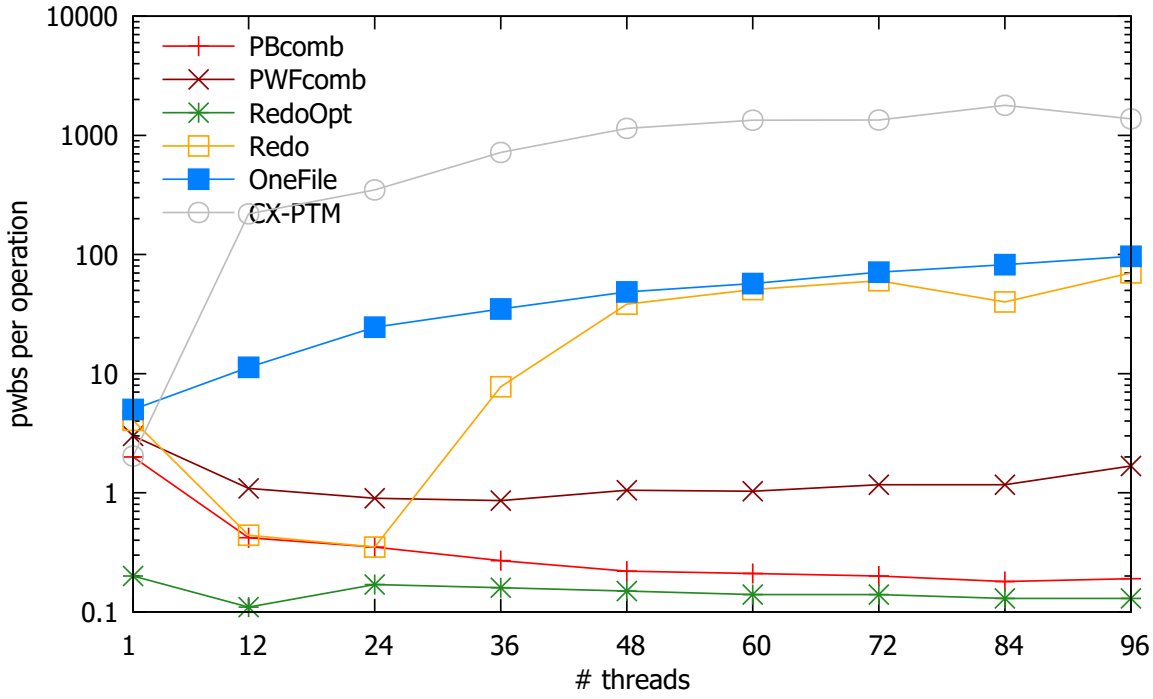


Figure 2: Simulation of a persistent `AtomicFloat` object on Intel Xeon: `pwb` instructions per operation.

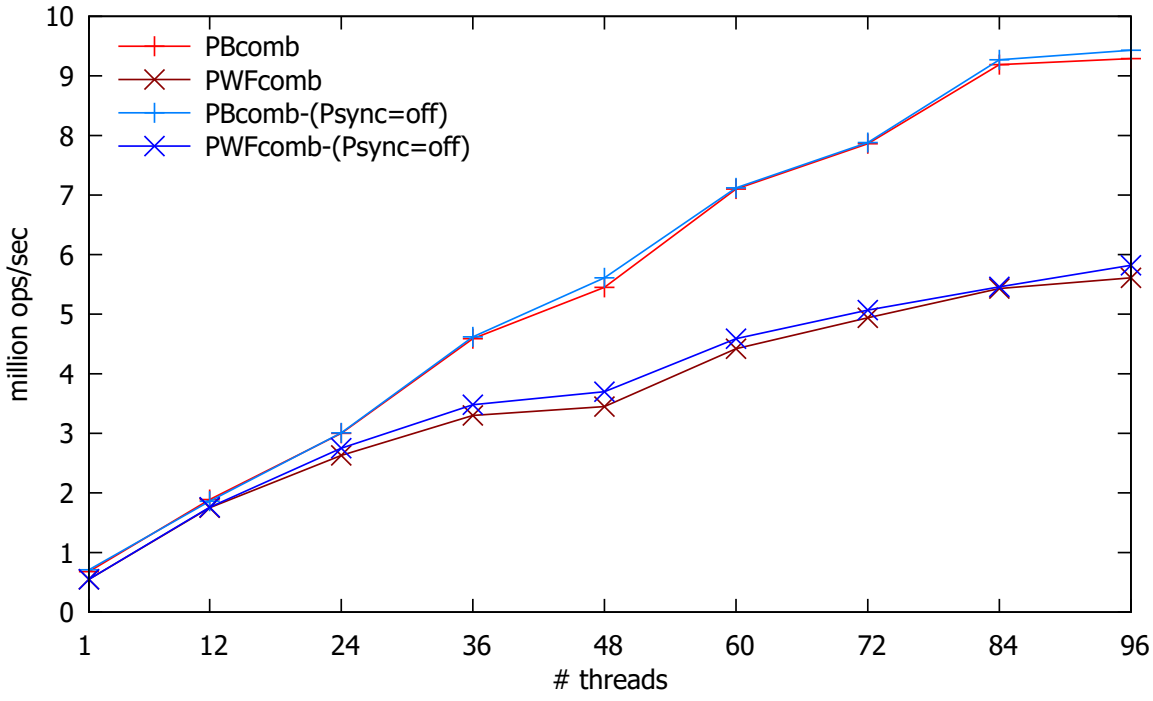


Figure 3: Simulation of a persistent `AtomicFloat` object on Intel Xeon: throughput with no `psync` instructions.

ent queue implementations; trends are similar to Figure 2. In Figure 6, we have replaced the `pwb` instructions with simple NOP operations and we measure the throughput of the different algorithms. The figure shows that the synchronization cost of PWFcomb and PBcomb is

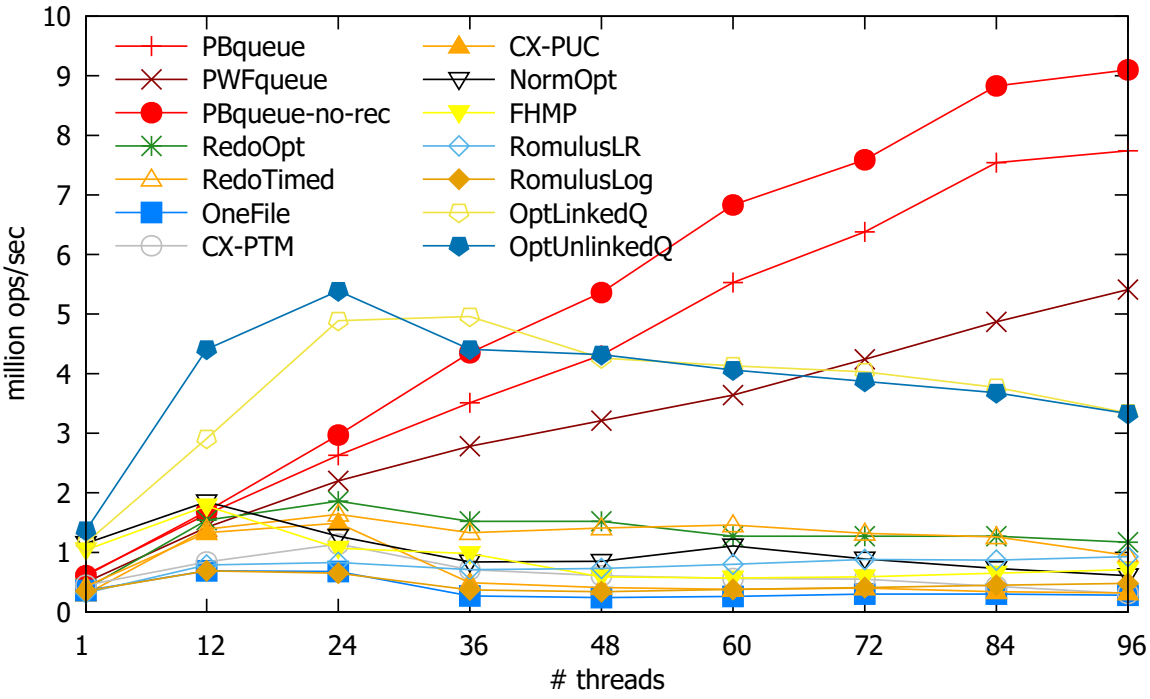


Figure 4: Persistent queue implementations on Intel Xeon: throughput.

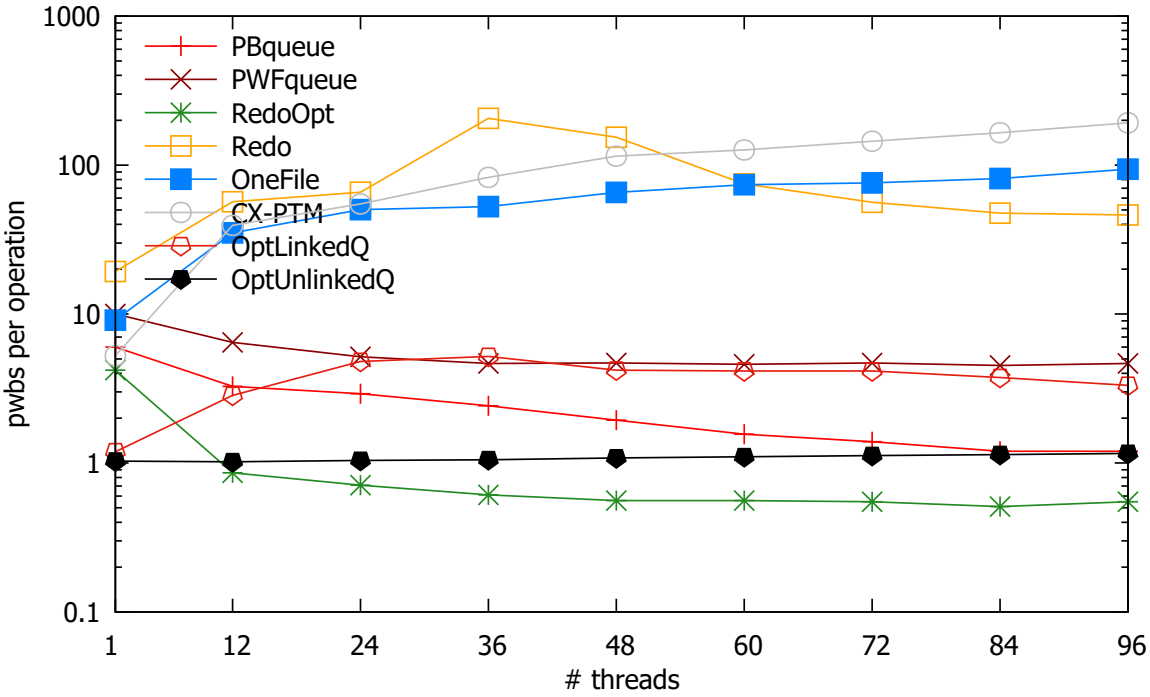


Figure 5: Persistent queue implementations on Intel Xeon: `pwb` instructions per operation.

much lower compared to its competitors. A comparison of Figure 5 with Figure 6 shows the performance impact of persistence.

Persistent Stacks. Figure 7a illustrates that the performance of PBstack and PWFstack is much better than the following algorithms: the persistent stack implementations based on OneFile [45] and Romulus [17], and a persistent stack based on flat-combining (DFC) [47],

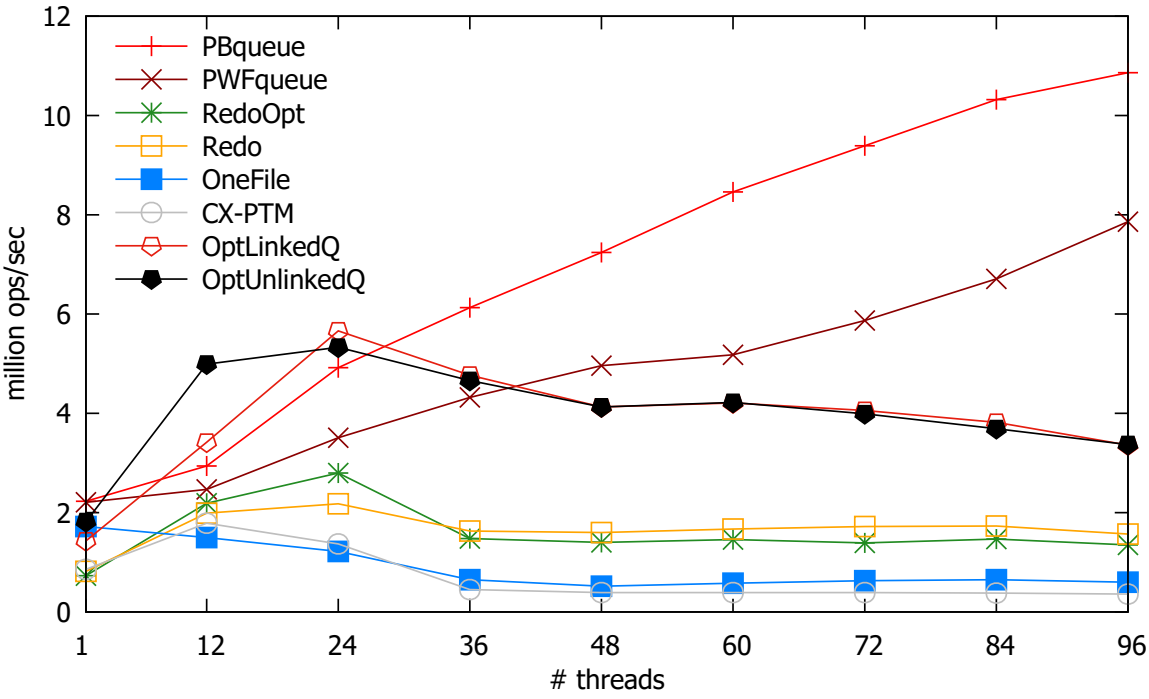


Figure 6: Persistent queue implementations on Intel Xeon: throughput with no `pwb` instructions.

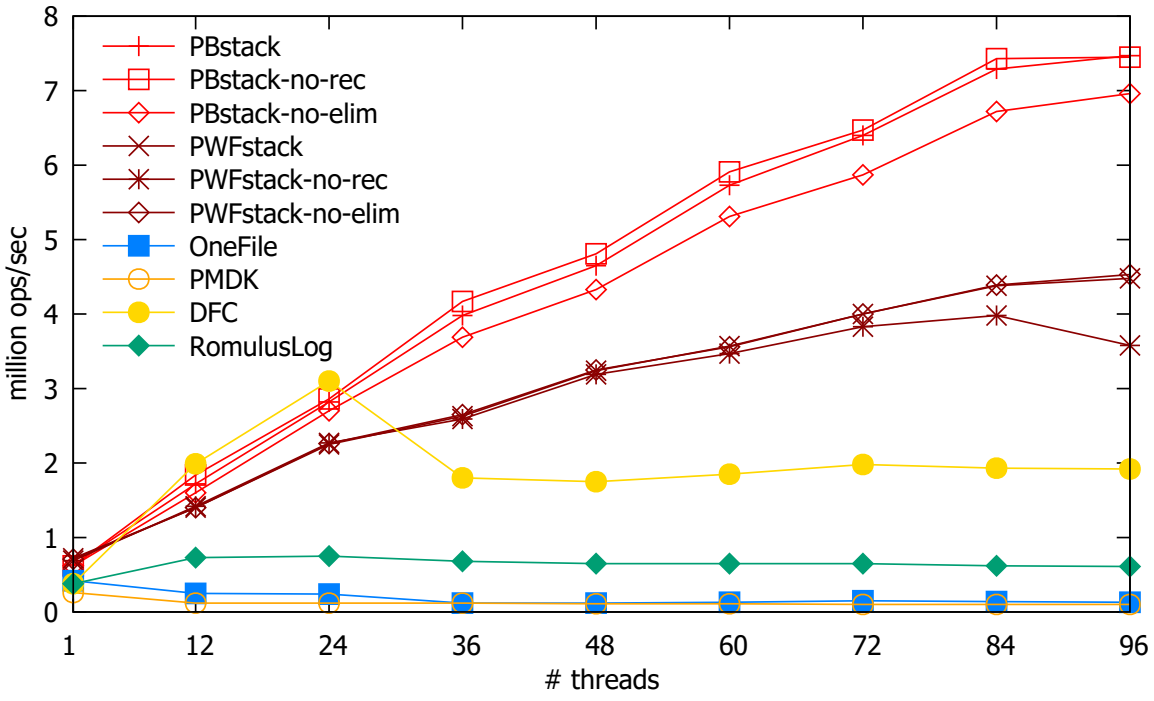
which is the best competitor.

Similarly to our stack implementations, DFC uses an announce array where threads can announce their requests. In contrast to our algorithms, DFC does not avoid the cost of persisting this array. DFC has each thread persisting its own element in the announce array. To ensure durable linearizability, a combiner serves only those requests whose announcements have been persisted. This requires an additional mechanism in order for a thread to inform the combiner that it has persisted its announcement. Another major difference of DFC from our approach is that in DFC the combiners perform updates directly on the state of the object. This introduces several difficulties for achieving persistence when designing the stack. Finally, DFC stores the return value for each thread in the announce array. This requires that the combiner persists each return value separately. These design decisions result in high persistence cost and synchronization overhead, as reflected in the figure.

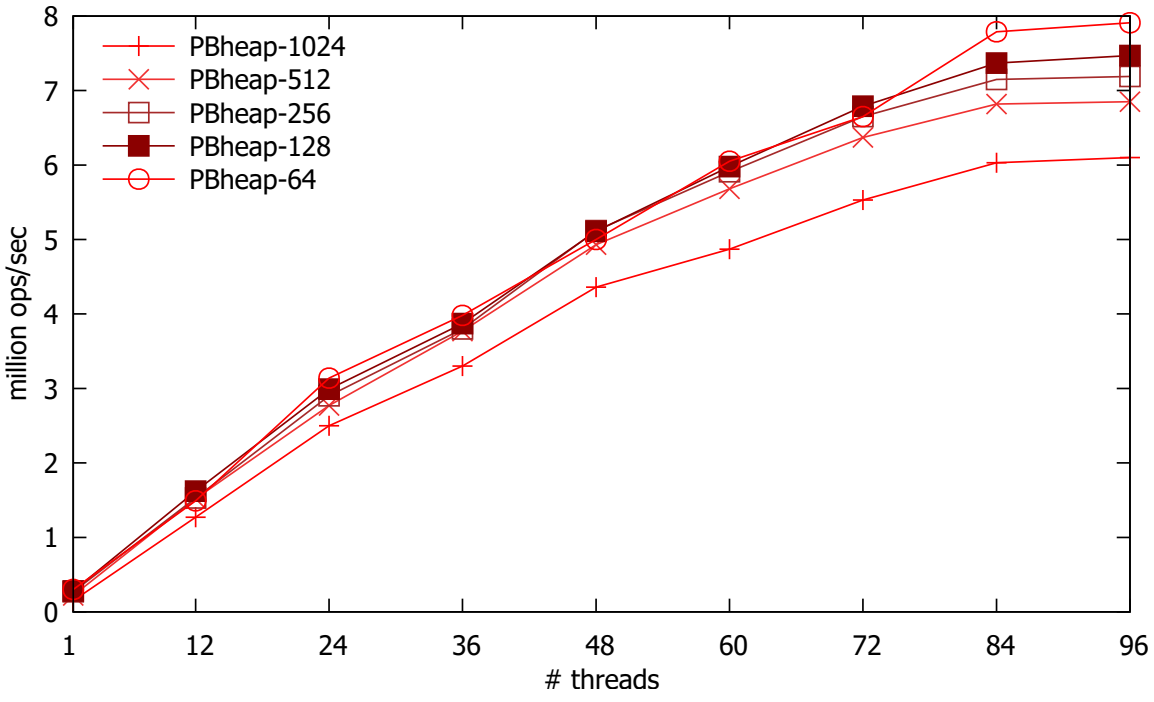
DFC applies elimination for reducing its persistence cost. However, the DFC design decision of performing updates directly on the shared state complicates its elimination scheme and its recovery code. We also applied elimination to our algorithms. Figure 7a (comparing the diagrams PBstack and PWFstack with PBstack-no-elim and PWFstack-no-elim, respectively) shows the positive impact of elimination in our stack implementations. As our implementations apply updates on copies of the state, the positive impact stems mainly from reducing their persistence cost (e.g., the number of newly allocated nodes that need to be persisted).

Memory Management. Diagrams PBstack-no-rec and PWFstack-no-rec, in Figure 7a, illustrate the impact of removing the scheme for recycling list nodes in our stacks. Comparing them with PBstack and PWFstack shows that our memory management scheme for stacks is very efficient. On the contrary, Figure 4 shows that the performance of PBqueue is negatively affected by the simple recycling scheme for nodes we apply in this case (Section 5).

Persistent Heaps. Figure 7b shows the throughput of PBheap for small and medium heap sizes (i.e., $64 - 1024$ keys). Initially, the heap is half-full. To make the experiment realistic, we avoid to have a full (or empty) heap by performing an equal number of HInsert and



(a)



(b)

Figure 7: Experiments on Intel Xeon: (a) throughput of persistent stack implementations and (b) throughput of persistent heap implementations based on PBcomb for different heap-sizes (64 - 1024).

HDeleteMin operations. Figure 7b shows that even for heaps of medium size, the performance of PBheap is good, illustrating that more complex persistent data-structures than stacks and queues can easily be implemented on top of our algorithms, and perform well when their size is not very large.

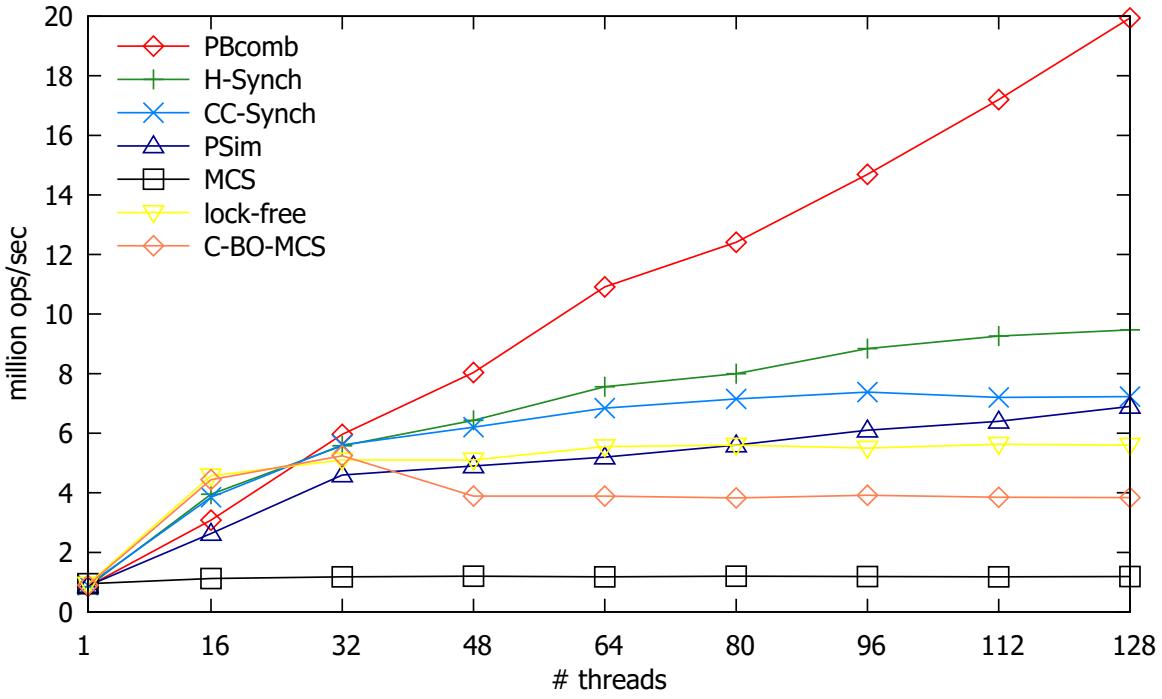


Figure 8: Throughput of implementations while simulating a volatile `AtomicFloat` object on AMD Epyc.

(per operation)	BCOMB	H-SYNCH	CC-SYNCH	PSIM
cache-misses	2.8	6.3	5.5	10
stores on cache-lines in shared state	0.0012	1.0	1.0	1.0
reads on cache-lines in shared state	0.034	1.8	1.6	1.3

Table 1: Performance counters using *Perf* for 128 threads.

Performance in systems with volatile memory. We study the performance of the volatile version of PBCOMB in a system without NVMM: a 64-core AMD Epyc consisting of 2 Epyc 7501 processors, which provides 64 cores (8 NUMA nodes, 128 logical cores); we saw similar performance behavior on the Intel Xeon machine. In Figure 8, the synthetic benchmark runs using H-SYNCH [22, 36], CC-SYNCH [22, 36], PSIM [21, 36], MCS queue spin-locks [40], a simple lock-free implementation [21, 23], and an hierarchical lock (C-BO-MCS) [20]. Figure 8 shows that a volatile version of PBCOMB exhibits much better performance than all other algorithms. In Table 1, we present results for 1) cache-misses per operation, 2) stores on cache-shared locations per operation, and 3) reads on cache-shared locations per operation. (More experiments are provided in [25].)

7 Related Work

A lot of work has been devoted to design persistent transactional memory systems (e.g., [53, 14, 13, 44, 34, 45, 8, 55, 39]). Such systems often rely on some kind of logging technique employing either redo logs [53, 34, 45] or undo logs [13, 44, 14]. Logging causes serious performance penalties as the log is usually stored in persistent memory. Our algorithms avoid logging to reduce both synchronization and persistence cost.

PMDK [44] attempts to reduce the logging cost by aggregating all updates performed on an object in a single transaction. Romulus [17] follows a different approach for achieving the same goal. Romulus comes in two flavors, RomulusLog which is blocking, and RomulusLR which

supports wait-free read-only transactions (and blocking update transactions).

OneFile [45] is a redo-log based persistent transactional system whose main characteristic is that its transactions do not maintain read-sets. However, it serializes all update transactions and all transactions (read-only and update) have to help update transactions to complete. OneFile comes in two versions, one lock-free and another wait-free. The wait-free version shares some ideas with PSim, thus integrating some form of combining, but it inherits the helping and logging mechanisms from the lock-free version. ONLL [15] is a log-based persistent universal construction which ensures durable linearizability [35] and lock-freedom. ONLL performs one persistent fence for each update operation and avoids performing persistence fences for read operations.

A persistent wait-free universal construction (CX-PUC) and a persistent transactional memory system (CX-PTM) are presented in [18]. Both algorithms are based on the universal construction provided in [19]. The algorithms store $2n$ replicas of the data structure in NVMM, and use a shared queue, stored in volatile memory, to impose an order to the executed operations. Threads synchronize using consensus objects in order to decide the order in which the operations will be applied on the data structure. A thread chooses one of the persistent copies of the data structure to work on and may require to execute all operations that precede its operation in the queue, in order to ensure consistency.

RedoOpt, presented also in [18], is a persistent, durably linearizable, wait-free universal construction that uses ideas from PSim to achieve lower persistence cost and better performance than CX-PUC and CX-PTM. RedoOpt employs the shared queue used by CX-PUC and CX-PTM, and therefore it does not avoid the synchronization overheads of them.

All these algorithms satisfy weaker consistency than detectable recoverability ensured by PBcomb and PWFcomb.

Capsules [10] can be used to transform concurrent algorithms that use only read and *CAS* primitives to their persistent versions. The programmer has to partition the code into parts, called *capsules*, each containing a single *CAS*. This *CAS* has to be replaced with its recoverable version [6]. We use an optimized version of Capsules, which can be applied only to *normalized* implementations [51], to our experiments, as it achieved better performance. Recent generic approaches for designing lock-free data structures appear in [27, 29]; they are not detectable recoverable and they do not experiment with stacks and queues.

The first hand-tuned durable queues were provided in [28]. One of them, namely the *log-queue*, ensured detectable recoverability, whereas the other two guaranteed *durable linearizability* [35] and *buffered durable linearizability* [35], respectively. Their design is based on the lock-free queue (MSQueue) presented by Michael and Scott [41]. A recent paper [50] presents hand-tuned durably linearizable queue implementations that outperform those in [28] and other previous persistent queue implementations. These implementations are designed based on the observation that minimizing accesses to flushed content could be beneficial for performance. Our experiments show that PBqueue outperforms the queues in [50] as the number of threads increases.

PBcomb and PWFcomb borrow and extend ideas from PSim [21, 23], a state-of-the-art wait-free practical software combining protocol, which is built upon the simple idea presented by Herlihy in [33]. A thread p first announces its request and informs other threads that it has an active request by applying a Fetch&dAdd instruction on an integer variable that implements a bit vector. Next, it finds out which other requests are active by reading this integer variable, and applies these requests to a local copy of the simulated object. Finally, it tries to change the shared pointer to the simulated object's state to point to this local copy. Similarly, PWFqueue is the persistent version of SimQueue. SimQueue allows the enqueuers and dequeuers to run independently by employing two instances of PSim. It also employs a linked list that is comprised of two parts for implementing the queue and have all threads performing appropriate actions to link these parts before serving requests.

All detectable algorithms we are aware of assume some system support to ensure detectability. Those in [9, 47, 5, 3] assume that for every thread p, the system calls the recovery function of the request req that p was executing at crash time, with the same arguments as req. We follow the same assumption in this paper. They also assume that p has a non-volatile private variable that recoverable operations and recovery functions use for managing check-points in their execution flow; the system sets the value of this variable to *0* just before p initiates the execution of a new request. Instead, we assume that p has a toggle bit which the system toggles each time p invokes a request and passes it as a parameter to the request (recall that we implement this mechanism through the use of *seq*). Our algorithms can be adjusted to work using check-pointing variables, as in [9, 47, 5, 3]. This may require to persist private non-volatile variables for each thread, which is expected to be of low cost [5]. The detectable algorithms in [28, 10] assume, as here, the use of a sequence number which is passed to recoverable operations via their arguments. Other detectable algorithms [6] also assume that the system persists some of the threads' state. Ben-Baruch *et al.* [9] prove that detectability cannot be achieved without system support. Specifically, they prove that for a specific class of objects (which include FIFO queues, considered in this paper), any *obstruction-free* detectable implementation must receive auxiliary state.

For our experiments, we tested code which is publicly available [36, 16, 49, 46], and we focus on persistent synchronization techniques, transactional memory systems and universal constructions, whose experimental platforms provide persistent stack and queue implementations.

8 Discussion

We present PBcomb and PWFcomb, highly-efficient recoverable software combining protocols that are many times faster than state-of-the-art recoverable universal constructions and software transactional systems. We identify three persistence principles, crucial for performance, and we illustrate how to make the appropriate design decisions to respect them when designing recoverable software combining protocols. Both PBcomb and PWFcomb can be used to derive recoverable implementations of any data structure from its sequential implementation. Thus, it is possible to develop a software-combining API that automatically transforms any data structure to fit our schemes by using a single instance of the corresponding algorithm. Our recoverable implementations of stacks, and the heap implementation, indeed follow this approach, using a single instance of PBcomb or PWFcomb. To increase parallelism and achieve better performance, PWFqueue (and PBqueue) employs a similar approach as SimQueue [21, 23] utilizing two instances of PWFcomb (PBcomb, respectively). Although this choice is not fundamentally necessary and made the queue implementations more complicated than using a single instance, it results in superior performance.

Coming up with a wait-free recoverable heap using PWFcomb is a relatively easy task. We are currently working on this direction, as well as on implementing a simple garbage collection scheme for PWFqueue. We will include the resulting algorithms in future versions of our library.

Software combining restricts parallelism by executing sequentially all requests. Thus, PBcomb and PWFcomb, although applicable, are not necessarily the best choices for implementing e.g., recoverable tree-like data structures, where threads may work on different subtrees without interference. Experiments for PBheap illustrate that PBcomb and PWFcomb may perform well in this case, only if the data structure size is small or medium. In [5], we present a generic approach for obtaining efficient recoverable such data structures, independently of their size, from their concurrent implementations.

In [26], more than one instance of PSim is used to efficiently implement an extendible hashing scheme. Using more instances of PBcomb and PWFcomb for efficiently implementing recoverable hashing, or recoverable tree-like data structures is an interesting open problem.

The performance of state-of-the-art combining protocols [22, 31] is still far from the ideal [24]; the ideal performance is measured in [24] by calculating the time that it takes to a single thread to execute the total number of synchronization requests (sidestepping the synchronization protocol) and perform the total amount of local work that follows its own synchronization requests. [24] proposes a technique, called Osci, that enables batching of the synchronization requests initiated by threads running on the same (oversubscribed) core. It studies the impact on performance of this technique, when it is combined with cheap context switching and shows that it is remarkable. Osci has performance which is very close to the ideal. Klaftenegger *et al.* [37] proposes a technique, similar to *futures* [7], where a thread does not block waiting the combiner to serve its request; it rather executes subsequent computation and may block when it needs to access some of the variables that are updated by the request. This technique increases parallelism and enhances performance. The paper [37] also focuses on the case where some of the requests do not require any response and shows that avoiding recording of responses could have a positive impact on performance. Examining whether the techniques presented in [24, 37] can be extended and combined with our results to get more efficient recoverable protocols is a potential path for future work.

A collection of arguments to support correctness of our protocols are provided in [25]. Using model checking or verification techniques for further checking correctness [12] would be a valid path to consider.

Acknowledgments. This research is supported by the EU Horizon 2020, Marie Sklodowska-Curie project with GA No 101031688, and by the Hellenic Foundation for Research and Innovation (HFRI) under the "Second Call for HFRI Research Projects to support Faculty members and researchers" (project number: 3684). For Eleftherios Kosmas, the 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).

References

[1] S. N. Agathos, N. D. Kallimanis, and V. V. Dimakopoulos. Speeding up openmp tasking. In *European Conference on Parallel Processing*, pages 650–661. Springer, 2012.

[2] A. Amer, C. Archer, M. Blocksome, C. Cao, M. Chuvelev, H. Fujita, M. Garzaran, Y. Guo, J. R. Hammond, S. Iwasaki, et al. Software combining to mitigate multithreaded mpi contention. In *Proceedings of the ACM International Conference on Supercomputing*, pages 367–379, 2019.

[3] H. Attiya, O. Ben-Baruch, P. Fatourou, D. Hendler, and E. Kosmas. Tracking in order to recover - detectable recovery of lock-free data structures. SPAA '20, pages 503–505, New York, NY, USA, 2020. Association for Computing Machinery.

[4] H. Attiya, O. Ben-Baruch, P. Fatourou, D. Hendler, and E. Kosmas. Tracking in order to recover: Recoverable lock-free data structures. *CoRR*, abs/1905.13600, 2021.

[5] H. Attiya, O. Ben-Baruch, P. Fatourou, D. Hendler, and E. Kosmas. Detectable recovery of lock-free data structures. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page to appear, New York, NY, USA, 2022. Association for Computing Machinery.

[6] H. Attiya, O. Ben-Baruch, and D. Hendler. 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, pages 7–16, 2018.

[7] H. C. Baker and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, page 55–59, New York, NY, USA, 1977. Association for Computing Machinery.

[8] H. A. Beadle, W. Cai, H. Wen, and M. L. Scott. Nonblocking persistent software transactional memory. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, pages 429–430, New York, NY, USA, 2020. Association for Computing Machinery.

[9] O. Ben-Baruch, D. Hendler, and M. Rusanovsky. Upper and lower bounds on the space complexity of detectable objects. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, pages 11–20, New York, NY, USA, 2020. Association for Computing Machinery.

[10] N. Ben-David, G. E. Blelloch, M. Friedman, and Y. Wei. Delay-free concurrency on faulty persistent memory. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, pages 253–264, New York, NY, USA, 2019. Association for Computing Machinery.

[11] G. E. Blelloch and Y. Wei. Brief Announcement: Concurrent Fixed-Size Allocation and Free in Constant Time. In H. Attiya, editor, *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 51:1–51:3, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[12] A. Bouajjani, M. Emmi, C. Enea, and S. Mutluergil. Proving linearizability using forward simulations. In *In: Majumdar R., Kuncak V. (eds) Computer Aided Verification.*, volume 10427 of *CAV '17*, New York, NY, USA, 2017. Lecture Notes in Computer Science, Springer, Cham.

[13] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *SIGPLAN Not.*, 49(10):433–452, Oct. 2014.

[14] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *SIGARCH Comput. Archit. News*, 39(1):105–118, Mar. 2011.

[15] N. Cohen, R. Guerraoui, and I. Zablotchi. The inherent cost of remembering consistently. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 259–269, New York, NY, USA, 2018. Association for Computing Machinery.

[16] A. Correia, P. Felber, and P. Ramalhete. The Code for RedoDB. https://github.com/pramalhe/RedoDB.

[17] A. Correia, P. Felber, and P. Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 271–282, New York, NY, USA, 2018. Association for Computing Machinery.

[18] A. Correia, P. Felber, and P. Ramalhete. Persistent memory and the rise of universal constructions. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[19] A. Correia, P. Ramalhete, and P. Felber. A wait-free universal construction for large objects. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, pages 102–116, New York, NY, USA, 2020. Association for Computing Machinery.

[20] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: a general technique for designing numa locks. *ACM SIGPLAN Notices*, 47(8):247–256, 2012.

[21] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 325–334, New York, NY, USA, 2011. Association for Computing Machinery.

[22] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–626, New York, NY, USA, 2012. Association for Computing Machinery.

[23] P. Fatourou and N. D. Kallimanis. Highly-efficient wait-free synchronization. *Theory of Computing Systems*, 55(3):475–520, 2014.

[24] P. Fatourou and N. D. Kallimanis. Lock oscillation: Boosting the performance of concurrent data structures. In *21st International Conference on Principles of Distributed Systems (OPODIS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[25] P. Fatourou, N. D. Kallimanis, and E. Kosmas. Persistent software combining. *CoRR*, abs/2107.03492, 2021.

[26] P. Fatourou, N. D. Kallimanis, and T. Ropars. An efficient wait-free resizable hash table. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, page 111–120, New York, NY, USA, 2018. Association for Computing Machinery.

[27] 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 of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 377–392, New York, NY, USA, 2020. Association for Computing Machinery.

[28] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank. A persistent lock-free queue for non-volatile memory. *ACM SIGPLAN Notices*, 53(1):28–40, 2018.

[29] M. Friedman, E. Petrank, and P. Ramalhete. Mirror: Making lock-free data structures persistent. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 1218–1232, New York, NY, USA, 2021. Association for Computing Machinery.

[30] R. L. Graham, T. S. Woodall, and J. M. Squyres. Open mpi: A flexible high performance mpi. In *Parallel Processing and Applied Mathematics*, pages 228–239, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[31] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364, 2010.

[32] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 206–215, New York, NY, USA, 2004. Association for Computing Machinery.

[33] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, nov 1993.

[34] J. Izraelevitz, T. Kelly, and A. Kolli. Failure-atomic persistent memory updates via justdo logging. *SIGPLAN Not.*, 51(4):427–442, Mar. 2016.

[35] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Proceedings of the 30th International Symposium of Distributed Computing*, volume LNCS 9888 of *DISC '16*, pages 313–327. Springer, 2016.

[36] N. D. Kallimanis. Synch: A framework for concurrent data-structures and benchmarks. https://github.com/nkallima/sim-universal-construction.

[37] D. Klaftenegger, K. Sagonas, and K. Winblad. Queue delegation locking. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):687–704, 2018.

[38] N. Li and W. Golab. Brief announcement: Detectable sequential specifications for recoverable shared objects. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 557–560, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. Ido: Compiler-directed failure atomicity for nonvolatile memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 258–270. IEEE Press, 2018.

[40] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[41] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

[42] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA '99)*, pages 182–204, 1999.

[43] M. Ploumidis, N. D. Kallimanis, M. Asiminakis, N. Chrysos, P. Xirouchakis, M. Gianoudis, L. Tzanakis, N. Dimou, A. Psistakis, P. Peristerakis, G. Kalokairinos, V. Papaefstathiou, and M. Katevenis. Software and hardware co-design for low-power hpc platforms. In *High Performance Computing*, pages 88–100. Springer International Publishing, 2019.

[44] PMDK. The persistent memory development kit. https://github.com/pmem/pmdk/.

[45] P. Ramalhete, A. Correia, P. Felber, and N. Cohen. Onefile: A wait-free persistent transactional memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 151–163. IEEE, 2019.

[46] M. Rusanovsky, O. Ben-Baruch, D. Hendler, and P. Ramalhete. The Code for DFC. https://github.com/matanr/detectable_flat_combining.

[47] M. Rusanovsky, O. Ben-Baruch, D. Hendler, and P. Ramalhete. A flat-combining-based persistent stack for non-volatile memory. *CoRR*, abs/2012.12868, 2020 (version submited at 23 December, 2020).

[48] T. R. Scogland and W.-c. Feng. Design and evaluation of scalable concurrent queues for many-core architectures. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, page 63–74, New York, NY, USA, 2015. Association for Computing Machinery.

[49] G. Sela and E. Petrank. The Code for Durable Queues. https://github.com/galysela/DurableQueues.

[50] G. Sela and E. Petrank. Durable queues: The second amendment. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 385–397, New York, NY, USA, 2021. Association for Computing Machinery.

[51] S. Timnat and E. Petrank. 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*, pages 357–368, 2014.

[52] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. 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*, pages 61–75, 2011.

[53] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *SIGPLAN Not.*, 46(3):91–104, Mar. 2011.

[54] K. Wu, J. Ren, I. Peng, and D. Li. Archtm: Architecture-aware, high performance transaction for persistent memory. pages 141–153, Feb. 2021.

[55] Y. Xu, J. Izraelevitz, and S. Swanson. Clobber-nvm: Log less, re-execute more. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, pages 346–359, New York, NY, USA, 2021. Association for Computing Machinery.

[56] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, Feb. 2020. USENIX Association.

type RequestRec {
    {ENQUEUE, DEQUEUE} *func*
    Integer *arg*
    Bit *activate*
    Bit *valid*
}
type Node {
    Data *data*
    Node **next*
}
type EStateRec {
    Node **Tail*
    ReturnValue $ReturnVal[0..n-1]$
    Bit $Deactivate[0..n-1]$
}
type DStateRec {
    Node **Head*
    ReturnValue $ReturnVal[0..n-1]$
    Bit $Deactivate[0..n-1]$
}

▷ Shared non-volatile variable:
Node $DUMMY$, initially $\langle \perp, \perp \rangle$

▷ Shared volatile variables:
Node **oldTail*, initially $\&DUMMY$
Set⟨Node*⟩ *toPersist*, initially $\emptyset$

▷ Shared non-volatiles variables used by the PBQUEUEENQ instance of PBCOMB:
EStateRec $EState[0..1]$, initially $\langle \&DUMMY, \langle \perp, \ldots, \perp \rangle, \langle 0, \ldots, 0 \rangle \rangle$
Bit $EIndex$, initially 0

▷ Shared volatile variables used by the PBQUEUEENQ instance of PBCOMB:
RequestRec $ERequest[0..n-1]$, initially $\langle \langle \perp, \perp, 0, 0 \rangle, \ldots, \langle \perp, \perp, 0, 0 \rangle \rangle$
Integer $ELock$, initially 0
Integer $ELockVal$, initially 0

▷ Shared non-volatile variables used by the PBQUEUEDEQ instance of PBCOMB:
DStateRec $DState[0..1]$, initially $\langle \&DUMMY, \langle \perp, \ldots, \perp \rangle, \langle 0, \ldots, 0 \rangle \rangle$
Bit $DIndex$, initially 0

▷ Shared volatile variable used by the PBQUEUEDEQ instance of PBCOMB:
RequestRec $DRequest[0..n-1]$, initially $\langle \langle \perp, \perp, 0, 0 \rangle, \ldots, \langle \perp, \perp, 0, 0 \rangle \rangle$
Integer $DLock$, initially 0
Integer $DLockVal$, initially 0

Figure 9: Types and initialization of PBQUEUE.

A Blocking Recoverable Queue

We present PBQUEUE, a blocking queue based on PBCOMB. PBQUEUE uses a singly linked list L to store the nodes of the queue. It employs two instances of PBCOMB, namely I_E and I_D, to synchronize the enqueuers and the dequeuers, respectively; this results in increased parallelism as it supports enqueues that are executed concurrently with dequeues. To achieve this, the first node of L plays the role of a dummy node.

A combiner in I_E serves only enqueue requests, while a combiner in I_D serves only dequeue requests. I_E stores just the tail of the queue in the *st* field of its *StateRec* records, whereas I_D stores just the head. We denote by *EStateRec* (*DStateRec*), *ERequest* (*DRequest*), *EState* (*DState*), and *EIndex* (*DIndex*) the *StateRec*, *Request*, *State*, and *MIndex* of PBCOMB used by I_E (I_D), as shown in Figure 9. Algorithms 5-6 provide the codes for the enqueuers, the dequeuers, and the standard sequential codes of the ENQUEUE and the DEQUEUE operations. The recovery function (which is similar to that of PBCOMB) in presented in Algorithm 7. Parts that differentiate from PBCOMB are highlighted in red and is responsible for persistence (e.g., it persists the nodes of L), as described below. The rest of the code resembles that of PBCOMB and it is easy to follow.

The persistence scheme of PBCOMB guarantees that the head and the tail of the simulated queue are persistent. In order to persist the changes performed on L, each enqueue combiner

Algorithm 5: PBQUEUE – Code of PBQUEUE and PBQUEUEENQ for thread $p \in \{0, \ldots, n-1\}$

```
   Procedure ReturnValue PBQueue(Function func, Data arg, Integer seq)
1      if func = Enqueue then PBQueueEnq(args, seq)
2      else PBQueueDeq(seq)
   Procedure PBQueueEnq(Data arg, Integer seq)
       // Announce request
3      ERequest[p] := ⟨Enqueue, Args, seq, 1 − ERequest[p].activate, 1⟩
4      return PerformEnqReq()
   Procedure ReturnValue PerformEnqReq()
       Bit ind                                                    // local variable of thread p
5      Integer lval                                               // local variable of thread p
6      while true do
7          lval := ELock
8          if lval mod 2 = 0 then
9              if CAS(ELock, lval, lval + 1) = true then break
10             lval := lval + 1
11         wait until ELock ≠ lval
12         if ERequest[p].activate = EState[EIndex].Deactivate[p] then
13             if ELockVal ≠ lval then wait until ELock ≠ lval + 2
14             return EState[EIndex].ReturnVal[p];
15     ind := 1 − EIndex
16     EState[ind] := EState[EIndex]                              // create a copy of current state
17     for q ← 0 to n − 1 do
           // if q has a request that is not yet Applied
18         if ERequest[q].activate ≠ EState[ind].Deactivate[q] and ERequest[q].valid = 1 then
19             add EState[ind].Tail to toPersist
20             Enqueue(&EState[ind].Tail, ERequest[q].arg)
21             EState[ind].ReturnVal[q] := ACK
22             EState[ind].Deactivate[q] := ERequest[q].activate
23     add EState[ind].Tail to toPersist
24     foreach e ∈ toPersist do pwb(e)
25     pwb(&EState[ind])
26     pfence()
27     ELockVal := ELock
28     EIndex := ind
29     pwb(&EIndex)
30     psync()
31     oldTail := EState[ind].Tail
32     toPersist = ∅
33     ELock := ELock + 1
34     return EState[EIndex].ReturnVal[p]
   Procedure Enqueue(Node **Tail, Data arg)
35     Node *newnode := allocate a new structure Node
36     newnode → data := arg
37     newnode → next := ⊥
38     ( *Tail) → next := newnode
39     *Tail := newnode
```

Algorithm 6: PBQUEUE – Code of PBQUEUEDEQ for thread $p \in \{0, \ldots, n-1\}$

```
    Procedure Node *PBQueueDeq(Integer seq)
        // Announce request
40      DRequest[p] := ⟨DEQUEUE, Args, seq, 1 − DRequest[p].activate, 1⟩⟩
41      return PerformRequest()
    Procedure ReturnValue PerformRequest()
42      Bit ind                                                  // local variables of thread p
43      Integer lval                                             // local variable of thread p
44      while true do
45          lval := DLock
46          if lval mod 2 = 0 then
47              if CAS(DLock, lval, lval + 1) = true then break
48              lval := lval + 1
49          wait until DLock ≠ lval
50          if DRequest[p].activate = DState[DIndex].Deactivate[p] then
51              if DLockVal ≠ lval then wait until DLock ≠ lval + 2
52              return DState[DIndex].ReturnVal[p];
53      ind := 1 − DIndex
54      DState[ind] := DState[DIndex]                            // create a copy of current state
        // Simulate the dequeues
55      for q ← 0 to n − 1 do
            // if q has a request that is not yet applied
56          if DRequest[q].activate ≠ DState[ind].Deactivate[q] and DRequest[q].valid = 1 then
57              if oldTail ≠ DState[ind].Head then
58                  returnVal := DEQUEUE(&DState[ind].Head)
59              else returnVal := ⊥
60              DState[ind].ReturnVal[q] := returnVal
61              DState[ind].Deactivate[q] := DRequest[q].activate
62      pwb(&DState[ind])
63      pfence()
64      DLockVal := DLock
65      DIndex := ind
66      pwb(&DIndex)
67      psync()
68      DLock := DLock + 1
69      return DState[DIndex].ReturnVal[p]
    Procedure Node *DEQUEUE(Node **Head)
70      Node *ret := ( *Head) → next
71      if ret ≠ ⊥ then *Head := ( *Head) → next
72      return ret
```

Algorithm 7: PBQUEUE – Code of RECOVER for thread $p \in \{0, \ldots, n-1\}$

```
    Procedure ReturnValue Recover(Function func, Data arg, Integer seq)
73      lTail := EState[EIndex].Tail
74      CAS(&oldTail, &DUMMY, lTail)
75      if func = Enqueue then
76          ERequest[p] := ⟨func, args, seq mod 2, 1⟩
            // if request is not yet applied
77          if EState[EIndex].Deactivate[p] ≠ seq mod 2 then
78              return PerformEnqReq()
            // request is applied
79          return EState[EIndex].ReturnVal[p]
80      else
81          DRequest[p] := ⟨func, args, seq mod 2, 1⟩
            // if request is not yet applied
82          if DState[DIndex].Deactivate[p] ≠ seq mod 2 then
83              return PerformDeqReq()
            // request is applied
84          return DState[DIndex].ReturnVal[p]
```

maintains a set *toPersist* containing pointers to those nodes of L that have either been updated (specifically, its *next* pointer) or been created by the combiner (lines 19 and 23). After its simulation phase, a combiner persists its modifications on L by executing a `pwb` for each element of *toPersist* (line 24). These `pwb`s are necessary; otherwise, the modifications performed by the combiner on L will not survive after a crash, which may result in an inconsistent state and violate durable linearizability. (We allocate new nodes in contiguous memory regions, thus these `pwb`s persist as few cache lines as possible.)

Note that a DEQUEUE operation only updates the head of the simulated queue and does not modify the nodes of L. Therefore, the effects of a dequeue combiner on the simulated state of the queue are correctly persisted by PBQUEUE. However, there is the following subtlety that needs to be addressed, regarding the nodes of L that can be removed by the dequeue combiners. An enqueue combiner simulates the active ENQUEUE requests by directly modifying the nodes of L and persists these modifications later. Thus, if no care is taken, a dequeue combiner p_d may remove from L nodes that have been appended by an active enqueue combiner p_e. Then, the dequeue combiner may complete (and thus the corresponding served DEQUEUE operations may respond) before the enqueue combiner persists its modifications. This jeopardizes detectability. Specifically, assume that p_d has removed a node containing data d while simulating a DEQUEUE request D, which has been appended in the list by p_e while simulating an ENQUEUE(d) request E. The process that announced D responds with d before p_e persists its modifications while simulating E. If a crash occurs, during recovery, the simulated state of the queue will not contain the response of E and thus E is considered not applied, after the crash. So, after the crash, it is like E has never enqueued d, since otherwise, it should be able to find its response. However, D has already responded with d before the crash. Durable linearizability still holds but detectability is violated.

To address this, PBQUEUE disallows dequeue combiners to remove any node of L that has not yet been persisted. It achieves this by using a shared volatile variable $oldTail$, which initially points to $DUMMY$. Just before releasing the lock, an enqueue combiner sets $oldTail$ to point to the last node of L (line 31). Then, the dequeue combiner removes nodes from L as long as $oldTail$ is not reached (lines 57-58). When $oldTail$ is reached, the dequeue combiner reports that the queue is empty for the corresponding DEQUEUE requests (line 59).

B Wait-Free Recoverable Queue

PWFQUEUE borrows ideas from SIMQUEUE [21, 23], so the description below follows that in [21, 23]. PWFQUEUE uses two instances of PWFCOMB. The first is used to synchronize the enqueuers and the second to synchronize the dequeuers. To achieve independent execution of the enqueuers from the dequeuers, the first node in the list is always considered as a dummy node. We denote by $EStateRec$, $ERequest$, $EState$, ES, $DFlush$, and $DCombRound$ the instances of $StateRec$, $Request$, $State$, S, $Flush$, and $CombRound$ of PWFCOMB used by the enqueuers. Similarly, we denote by $DStateRec$, $DRequest$, $DState$, DS, $DFlush$, and $DCombRound$ the instances of $StateRec$, $Request$, $State$, S, $Flush$, and $CombRound$ of PWFCOMB used by the dequeuers. The queue is implemented by a singly-linked list, where $DS \rightarrow Head$ points to the head node of the queue and $ES \rightarrow Tail$ points to the tail node of the queue.

Algorithm 9 provides the code for the enqueuers, where parts that differentiate from PWFCOMB are identified in red. A thread p executing an enqueue request will also try to serve enqueue requests by other enqueuers. It does so by creating a local list of new nodes that will eventually be appended to the current state of the queue. Note that $EStateRec$ contains not only a pointer, $oldTail$, to the element reached by following *next* pointers starting from the head of the queue, but also two pointers, called $lHead$ and $Tail$, pointing to the first and last elements, respectively, of the local list created by the last combiner that updated ES. We remark that at every point in time, the state of the queue is comprised of all nodes (other than

type RequestRec {
    {ENQUEUE, DEQUEUE} *func*
    Integer *arg*
    Bit *activate*
    Bit *valid*
}
type Node {
    Data *data*
    Node **next*
}
type EStateRec {
    Node **Tail*
    Node **oldTail*
    Node **lhead*
    ReturnValue $ReturnVal[0..n-1]$
    Bit $Deactivate[0..n-1]$
    Bit $Index[0..n-1]$
    $\{0, 1, .., n-1\}$ *pid*
}
type DStateRec {
    Node **Head*
    ReturnValue $ReturnVal[0..n-1]$
    Bit $Deactivate[0..n-1]$
    Bit $Index[0..n-1]$
    $\{0, 1, .., n-1\}$ *pid*
}

▷ Local volatile variable:
Set⟨Node*⟩ *newItems*

▷ Shared non-volatile variable:
Node $DUMMY$, initially $\langle \perp, \perp \rangle$

▷ Shared non-volatile variables used by the PWFENQUEUE instance of PWFCOMB:
EStateRec $EState[0..n][0..1]$, initially $\langle \&DUMMY, \perp, \perp, \perp, \langle \perp, \ldots, \perp \rangle, \langle 0, \ldots, 0 \rangle, \langle 0, \ldots, 0 \rangle, 0 \rangle$
EStateRec $*ES := \&EState[n][0]$, initially ES points to $EState[n][0]$

▷ Shared volatile variables used by the PWFENQUEUE instance of PWFCOMB:
RequestRec $ERequest[0..n-1]$, initially $\langle \langle \perp, \perp, 0, 0 \rangle, \ldots, \langle \perp, \perp, 0, 0 \rangle \rangle$
Integer $EFlush[0..n-1]$, initially $\langle 0, \ldots, 0 \rangle$
Integer $ECombRound[0..n-1][0..n-1]$, initially $\langle \langle 0, \ldots, 0 \rangle, \ldots, \langle 0, \ldots, 0 \rangle \rangle$

▷ Shared non-volatile variables used by the PWFDEQUEUE instance of PWFCOMB:
DStateRec $DState[0..n][0..1]$, initially $\langle \&DUMMY, \langle \perp, \ldots, \perp \rangle, \langle 0, \ldots, 0 \rangle, \langle 0, \ldots, 0 \rangle, 0 \rangle$
DStateRec $*DS := \&DState[n][0]$, initially DS points to $DState[n][0]$

▷ Shared volatile variables used by the PWFENQUEUE instance of PWFCOMB:
RequestRec $DRequest[0..n-1]$, initially $\langle \langle \perp, \perp, 0, 0 \rangle, \ldots, \langle \perp, \perp, 0, 0 \rangle \rangle$
Integer $DFlush[0..n-1]$, initially $\langle 0, \ldots, 0 \rangle$
Integer $DCombRound[0..n-1][0..n-1]$, initially $\langle \langle 0, \ldots, 0 \rangle, \ldots, \langle 0, \ldots, 0 \rangle \rangle$

Figure 10: Types and initialization of PWFQUEUE.

the dummy) that can be reached by following next pointers starting from the node pointed to by $DS \rightarrow Head$, as well as of all nodes that can be reached by following the *next* pointers starting from the node pointed to by $ES \rightarrow lHead$. Thus, $ES \rightarrow oldTail$ points to the node that was the tail of the queue before the last SC on ES was performed, $ES \rightarrow Tail$ points to the current tail node of the queue, and $lHead$ points to a node that will end up to be the node pointed to by $ES \rightarrow oldTail \rightarrow next$. The linking of $ES \rightarrow oldTail \rightarrow next$ to point to $ES \rightarrow lHead$ is performed by all enqueuers, by calling ENQUEUECONNECT (line 33), before they proceed to serve any request.

Algorithm 10 provides the code for the dequeuers. The only subttlety is that a dequeuer should also ensure that the entire queue (which is now comprised of two lists) is connected before it proceeds to serve the dequeues, as otherwise consistency may be jeopardized. This is done by calling DEQUEUECONNECT (line 88). The rest of the code resembles that of PWFCOMB and is therefore easy to follow.

The persistence scheme of PWFQUEUE follows that of PWFCOMB. The persistence of PWFCOMB guarantees that each of its two instances utilized in PWFQUEUE is persistent. The

Algorithm 8: PWFQUEUE – Code of PWFQUEUE, PWFENQUEUE, PWFDEQUEUE, and RECOVER for process $p \in \{0, \ldots, n-1\}$

```
Procedure ReturnValue PWFQueue(Function func, Data arg, Integer seq)
1  if func = Enqueue then PWFenqueue(arg, seq)
2  else PWFdequeue(seq)

Procedure PWFenqueue(Data arg, Integer seq)
   // Announce request
3  ERequest[p] := ⟨Enqueue, arg, seq, 1 − ERequest[p].activate, 1⟩
4  Backoff()
5  return PerformEnqReq()

Procedure Node *PWFdequeue(Integer seq)
   // Announce request
6  DRequest[p] := ⟨Dequeue, ⊥, seq, 1 − DRequest[p].activate, 1⟩
7  Backoff()
8  return PerformDeqReq()

Procedure ReturnValue Recover(Function func, Data arg, Integer seq)
9  if func = Enqueue then
10 |  ERequest[p] := ⟨func, args, seq mod 2, 1⟩
   |  // if request is not yet applied
11 |  if ES → Deactivate[p] ≠ seq mod 2 then
12 |  |  Backoff()
13 |  |  return PerformEnqReq()
   |  // request is applied
14 |  return ES → ReturnVal[p].ret
15 else
16 |  DRequest[p] := ⟨func, args, seq mod 2, 1⟩
   |  // if request is not yet applied
17 |  if DS → Deactivate[p] ≠ seq mod 2 then
18 |  |  Backoff()
19 |  |  return PerformDeqReq()
   |  // request is applied
20 |  return DS → ReturnVal[p].ret
```

only subtlety comes from the necessity to connect the two parts of the linked list representing the queue. An enqueuer has to persist the new value of the *next* field of the node it updates in EnqueueConnect, for the following reason. Assume that two enqueue operations are executed by the same thread without any dequeuer to take steps in the mean-time. Thus, new nodes have been connected to the linked list implementing the queue twice. If a crash occurs, upon recovery, the three pointers stored in ES allows us to recover just the last of these connections (since ES is persisted on lines 48 and 56). This results in an inconsistent state.

Note that although dequeuers also help connecting the two parts of the list, they do not have to persist the *next* field of the node *nd* updated in DequeueConnect. Assume that a Dequeue operation has been persisted after traversing a link which has not been persisted yet. If the system fails, upon recovery, the head of the list has been persisted to point to a subsequent node to *nd* and thus *nd* will be never be accessed by any dequeuer again.

Algorithm 9: PWFQUEUE – Code of PERFORMENQREQ, ENQUEUE, and ENQUEUECONNECT for process $p \in \{0, \ldots, n-1\}$

Procedure ReturnValue PERFORMENQREQ()

```
21 EStateRec *lsPtr
22 Integer lval
23 for l ← 1 to 2 do
24     lsPtr := LL(ES)
25     Bit ind := lsPtr → Index[p]
26     EState[i][ind] := *lsPtr                                   // copy current state
27     EState[p][ind].pid := p
28     lval := EFlush[lsPtr → pid]
29     if lval mod 2 = 0 then lval := lval + 1
30     else lval := lval + 2
31     if VL(ES) = false then continue
32     if EState[p][ind].oldTail ≠ ⊥ and EState[p][ind].oldTail → next = ⊥ then
33         EnqueueConnect(&EState[p][ind])
34     for q ← 0 to n − 1 do
           // if q has a request that is not yet Applied
35         if ERequest[q].activate ≠ EState[p][ind].Deactivate[q] and ERequest[q].valid = 1 then
36             Enqueue(&EState[p][ind], ERequest[q].arg)
37             EState[p][ind].ReturnVal[q] := ACK
38             EState[p][ind].Deactivate[q] := ERequest[q].activate
39             ECombRound[p][q] := lval
40     if VL(ES) = true then
41         EState[p][ind].Index[p] := 1 − EState[p][ind].Index[p]
42         foreach e ∈ newItems do pwb(e)
43         pwb(&EState[p][ind])
44         pfence()
45         EFlush[p] := lval
46         if SC(ES, &EState[p][ind]) = true                      // Try to change the contents of ES
47          then
48             pwb(&ES)
49             psync()
50             CAS(&EFlush[p], lval, lval + 1)
51             return ES → ReturnVal[p]
52         BackoffCalculate();
53 lsPtr := ES
54 lval := EFlush[lsPtr → pid]
55 if lval mod 2 = 1 and lval = ECombRound[lsptr → pid][p] then
56     pwb(&ES)
57     psync()
58     CAS(EFlush[p], lval, lval + 1)
59 return ES → ReturnVal[p]
```

Procedure ENQUEUE(EStateRec *lS, Data arg)

```
60 Node *newnode := allocate a new structure Node
61 newnode → data := arg
62 newnode → next := ⊥
63 add newnode to newItems
64 if lS → oldTail = ⊥ then
65     lS → oldTail := lS → Tail
66     lS → oldTailNext := newnode
67 lS → Tail → next := newnode
68 lS → Tail := newnode
```

Procedure ENQUEUECONNECT(EStateRec *lS)

```
69 if lS → oldTail → next = ⊥ then
70     CAS(&(lS → oldTail → next), ⊥, Mark(lS → lhead))
71 if IsMarked(lS → oldTail → next) = true then
72     pwb(&(lS → oldTail → next))
73     lS → oldTail → next := UnMark(lS → oldTail → next)          // only for performance
```

Algorithm 10: PWFQUEUE – Code of PERFORMDEQREQ, DEQUEUE, and DEQUEUE-CONNECT for process $p \in \{0, \ldots, n-1\}$

Procedure ReturnValue PERFORMDEQREQ()

```
74  DStateRec *lsPtr
75  Integer lval
76  for l ← 1 to 2 do
77      lsPtr := LL(DS)
78      Bit ind := lsPtr → Index[p]
79      DState[i][ind] := *lsPtr                                   // copy current state
80      DState[p][ind].pid := p
81      lval := DFlush[lsPtr → pid]
82      if lval mod 2 = 0 then lval := lval + 1
83      else lval := lval + 2
84      if VL(DS) = false then continue
85      for q ← 0 to n − 1 do
            // if q has a request that is not yet Applied
86          if DRequest[q].activate ≠ DState[p][ind].Deactivate[q] and DRequest[q].valid = 1 then
87              if DState[p][ind].Head → next = ⊥ then
88                  DequeueConnect()
89              if DState[p][ind].Head → next ≠ ⊥ then
90                  returnVal := DEQUEUE(&DState[p][ind].Head)
91              else returnVal := ⊥
92              DState[p][ind].ReturnVal[q] := returnVal
93              DState[p][ind].Deactivate[q] := DRequest[q].activate DCombRound[p][q] := lval
94      if VL(DS) = true then
95          DState[p][ind].Index[p] := 1 − DState[p][ind].Index[p]
96          pwb(&DState[p][ind])
97          pfence()
98          DFlush[p] := lval
            // Try to change the contents of DS
99          if SC(DS, &DState[p][ind]) = true then
100             pwb(&DS)
101             psync()
102             CAS(&DFlush[p], lval, lval + 1)
103             return DS → ReturnVal[p]
104         BackoffCalculate();
105 lsPtr := DS
106 lval := DFlush[lsPtr → pid]
107 if lval mod 2 = 1 and lval = DCombRound[lsptr → pid][p] then
108     pwb(&DS)
109     psync()
110     CAS(DFlush[p], lval, lval + 1)
111 return DS → ReturnVal[p]
```

Procedure Node *DEQUEUE(Node **Head)

```
112 Node *ret := ( *Head) → next
113 if ISMARKED(ret) = true then
114     ret := UNMARK(ret)
115 *Head := ret
116 return ret
```

Procedure DEQUEUECONNECT()

```
117 EStateRec *lsPtr
118 Node *oldTail
119 Node *oldTailNext
120 lsPtr := LL(ES)
121 oldTail := lsPtr → oldTail
122 oldTailNext := lsPtr → oldTailNext
123 if VL(ES) and oldTail → next = ⊥ then
124     CAS(&(oldTail → next), ⊥, MARK(oldTailNext))
```