Recoverable and Detectable Fetch&Add

Liad Nahum
Department of Computer Science, Ben Gurion University, Beer Sheva, Israel

Hagit Attiya
Department of Computer Science, Technion, Haifa, Israel

Ohad Ben-Baruch
Department of Computer Science, Ben Gurion University, Beer Sheva, Israel

Danny Hendler
Department of Computer Science, Ben Gurion University, Beer Sheva, Israel

Abstract

The emergence of systems with non-volatile main memory (NVRAM) increases the need for persistent concurrent objects. Of specific interest are recoverable implementations that, in addition to being robust to crash-failures, are also *detectable*. Detectability ensures that upon recovery, it is possible to infer whether the failed operation took effect or not and, in the former case, obtain its response.

This work presents two recoverable detectable *Fetch&Add* (*FAA*) algorithms that are *self-implementations*, i.e, use only a *fetch&add* base object, in addition to read/write registers. The algorithms target two different models for recovery: the *global-crash* model and the *individual-crash* model. In both algorithms, operations are *wait-free* when there are no crashes, but the recovery code may block if there are repeated failures. We also prove that in the individual-crash model, there is no implementation of recoverable and detectable *FAA* using only read, write and *fetch&add* primitives in which all operations, including recovery, are lock-free.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms

Keywords and phrases Multi-core algorithms, persistent memory, non-volatile memory

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2021.29

Funding Supported by the Israel Science Foundation (grant number 380/18).

1 Introduction

Systems with byte-addressable *non-volatile main memory* (*NVRAM*) combine the performance benefits of conventional main memory with the durability of secondary storage. The emergence of commerical systems with NVRAM increased the interest in the *crash-recovery* model, in which failed processes may be resurrected after they crash. For this model, the goal is to design *recoverable concurrent objects* (also called *persistent* or *durable*): Objects that are made robust to crash-failures by allowing operations to recover from such failures.

Persistent objects were hand-crafted for specific data structures, e.g., [15,26,28,29]. Other work introduces general mechanisms to port existing algorithms and make them persistent, e.g., by using transactional memory [6, 9, 24, 27], universal constructions [5, 8, 10], or for specific families of algorithms [4,11,13]. These transformations rely on strong primitives such as *compare&swap*, while their non-persistent counterparts may use only weaker primitives, in terms of their level in the *consensus hierarchy* [21].

An alternative approach is to design persistent *self-implementations*, in which a recoverable operation is implemented by using non-recoverable instances *of the same primitive operation*, possibly with additional reads and writes on shared variables. Self-implementations can be used to implement high-level persistent objects by plugging them within existing object implementations. A recoverable implementations is *detectable* [15] if, in addition to being robust to crash-failures, it ensures that it is possible to infer, upon recovery, whether the failed operation took effect or not and, in the former case, obtain its response.

© Liad Nahum, Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler;
licensed under Creative Commons License CC-BY 4.0
25th International Conference on Principles of Distributed Systems (OPODIS 2021).
Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 29; pp. 29:1–29:17

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

For example, a detectable *compare&swap* (CAS) was used in a CAS-based generic transformation that makes algorithms recoverable [4]. Detectable self-implementations were presented in prior works for read, write, *test&set*, and CAS objects [2, 4]. An important primitive, which is useful in several data structures, is *fetch&add*, whose consensus number is two, i.e., it allows exactly two processes to solve consensus [21].

Our Contributions. This paper presents two detectable self-implementations of *Fetch&Add* (*FAA*). The first algorithm is for the *global-crash* model, where the whole system crashes and a single process is responsible for the recovery of all failed operations. The second algorithm is for the *individual-crash* model, where some processes may crash while others might not, and each process is responsible for restoring its own state in a consistent manner. However, recovering processes may have to wait for other processes to make progress with their operations or with their recovery code, and may never complete if such progress does not occur. We also prove that in the individual-crash model, there is no lock-free implementation of recoverable FAA objects from read, write and *fetch&add* primitives. In other words, for every detectable self-implementation of an FAA object in this model, either the *FAA* operation or the recovery code is not lock-free.

Our implementations satisfy *nesting-safe recoverable linearizability* (*NRL*) [2]. NRL was originally defined for the individual-crash model, and we extend it to the global-crash model. NRL implies that, following recovery, an implemented (higher-level) recoverable operation is able to complete its invocation of a base-object operation and obtain its response.

Related Work. The notion of detectability was presented in [14, 15]. A strict version of detectability, named *nesting-safe recoverable linearizability* (*NRL*), was formally defined by [2]. It requires that each process complete its operation and obtain its response before invoking another operation even when it incurs crash-failures. There are NRL self-implementations of recoverable *read*, *write*, *test&set* and *compare&swap* [2]. In a sense, FAA is a more complex object since, in most cases, an FAA operation has a unique place in history where it must be linearized, and its response is also unique based on this linearization point. Unlike FAA, in *compare&swap* and *test&set* we have more freedom in choosing where to linearize operations. For example, we can linearize a *test&set* operation that returns 1 at any point after the first operation in the linearization order. Tracking this unique linearization point of every crashed FAA operation and restoring the response based on it is the core challenge of our self-implementations. It is known that there is no wait-free self-implementation of a detectable *test&set* object [2]. Both this proof and our impossibility proof for FAA employ valency arguments that rely on the loss of response values incurred by processes following crash-failures.

Golab [16] defined *recoverable consensus* and revised the consensus hierarchy in the presence of crash-recovery failures, for both the individual-crash model and the global-crash model. (Recall that the *wait-free consensus hierarchy* [21] ranks shared objects according to the maximum number of processes that can use them to solve consensus; *fetch&add* and *test&set* are at level 2 of the hierarchy.) Golab showed that *test&set* drops to level 1 for the individual-crash model, if the number of crashes is unbounded. Our impossibility result can be adapted to prove an analogous result for *fetch&add*.

Other correctness conditions were suggested for shared objects that tolerate crash-recovery failures. *Strict linearizability* [1] treats the crash of a process as a response, either successful or unsuccessful, to the interrupted operation. *Persistent atomicity* [20] is similar to strict linearizability, but allows an operation interrupted by a failure to take effect before the next

invocation of the same process, possibly after the failure. Both conditions ensure that the state of an object is consistent after a crash. *Recoverable linearizability* [5] ensures object implementations can be composed, but may compromise program order following a crash. They also present a universal construction using *compare&swap* in the individual-crash model.

In the *recoverable mutual exclusion* problem (RME) [18], processes may undergo individual crashes during the execution of a mutual exclusion protocol. This paper also presents an RME algorithm using only reads and writes whose *remote memory references* (RMRs) complexity is logarithmic in the number of processes, n. There is an RME algorithm for the cache-coherent (CC) model [17], using *fetch&store* and *compare&swap*, which incurs $O(\frac{\log N}{\log\log N})$ RMRs. An RME algorithm with the same asymptotic RMR complexity was proposed for the distributed shared memory (DSM) and CC models [25]; it uses *fetch&store*. There is also an RME algorithm using only *compare&swap* and *fetch&increment*, with constant amortized RMR complexity [7]. In the global crash model, RME can be solved in a constant number of RMRs [19].

2 Model of Computation

We consider a system with N processes, $p_0, \ldots, p_{N-1}$, which communicate by applying atomic *primitive* operations (also called *primitives*) to *shared* base objects; the primitives applied are read, write and *fetch&add*. All shared base objects are non-volatile. The *state* of each process comprises its program counter and local variables; all local variables are volatile.[1] A *configuration* consists of the states of all processes and the values of all shared base objects. Two configurations C_1 and C_2 are *indistinguishable* to a set of processes P, denoted $C_1 \stackrel{P}{\sim} C_2$, if every process in P has the same state in C_1 and C_2, and all shared objects hold the same values in C_1 and C_2. The state of the system changes when processes take *steps*, each of which is a local computation followed by an atomic operation on one shared object (*ordinary* step), a *crash* step, or a *recovery* step.

Base objects and primitives are used to implement more complex objects, by specifying an algorithm for each operation of the implemented object using primitives on base objects. In this work, we implement a recoverable *Fetch&Add* (*FAA*) object that is detectable. The *sequential specification* of *fetch&add* contains all sequences of $FAA(v)$ operations in which each operation returns the sum of the arguments of all preceding *FAA* operations. We refer to the recoverable detectable operation that is implemented as *Fetch&Add*, while *fetch&add* is the primitive operation supported by the system, which can be applied to non-volatile variables.

An *execution* α is an alternating sequence of configurations and steps that follow the algorithm. An execution α is *crash-free* if it contains no crash steps, and hence, also no recovery steps. If a step s is possible in a configuration C at the end of a finite execution α, then the sequence obtained by appending s to α is also an execution, denoted $\alpha \circ s$, whose final configuration is denoted $C \circ s$. Let α be an execution ending in configuration C and let p be a process. If p's last step in α is a crash step, then the only step by p that is possible in C is a recovery step.

[1] We assume the simple mode of *shared caches*, where updates to the persistent shared base objects are immediate. There are standard ways to port algorithm from this model to more realistic models capturing existing architectures [23].

Process i invokes an operation *Op* on an object with an *invocation step*, and it *completes* with a *response step*, in which *Op*'s response is stored to a local (volatile) variable of process i. The return value is lost if process i crashes, unless process i writes it to a non-volatile variable before the crash, thereby *persisting* it. An operation *Op* is *pending* if it is invoked but not yet completed; each process has at most one operation pending.

We consider two models of recovery from crashes. In the *individual crash* model, at any point during an execution, each process can incur a crash that resets all its local variables to arbitrary values, but preserves the values of shared non-volatile variables. Recovery is done by the same crashed process, so each *recoverable operation*, *Op*, is associated with a *recovery function*, *Op.RECOVER*. In the *global crash* model, at any point during the execution, a global crash can occur that resets all local variables of all processes, but preserves the values of shared non-volatile variables. The recovery is done by a single process that is responsible for recovering all processes. In this case, we have a global *RECOVER* function; once *RECOVER* completes, processes can resume their execution.

One component of the processes' state is *Seq*[N]. For each process i, *Seq*[i] holds the sequence number of its current *FAA* operation. Before an *FAA* operation is invoked, *Seq*[i] is incremented by 1 by the process (or the system), externally to the operation itself. This is essential in our model for determining, upon recovery, the progress made by *FAA* operations before the crash. It has been proved [3] that detectable algorithms must keep auxiliary state, provided from outside the operation, either by the system or by the caller of the operation via arguments or a non-volatile variable accessible by them. This auxiliary state is used to infer where the failure occurred.

We say that an operation op_1 *precedes* another operation op_2 *in the real-time order* of an execution α, if op_1 completes before op_2 is invoked. Informally, a crash-free execution α is *linearizable* [22] if we can order all completed operations, as well as a subset of the pending operations, in a way that preserves the real-time order of the operations, and the return values respect the sequential specification of the *Fetch&Add* object.

An execution α satisfies *nesting-safe recoverable linearizability* (NRL) if the execution obtained by removing all crash and recovery steps from α is linearizable. NRL implies *detectability* [15], namely, a recovering operation has an appropriate response.

In our algorithms, the operations are *wait-free*, i.e., the execution of an operation by a process that does not incur a crash (global or individual) is guaranteed to complete in a finite number of its steps, regardless of the steps or crashes of other processes. An algorithm is *lock-free* if, whenever a set of processes take a sufficient number of steps and none of them crashes, then it is guaranteed that one of them will complete its operation.

3 FAA Implementation in the Global-Crash Model

Our algorithm implements a recoverable detectable *FAA* operation sing a *fetch&add* base object of unbounded size. An *FAA* operation receives, as its single argument, a value *val* that should be added to the global unbounded counter. *FAA* atomically adds *val* and returns the previous value of the counter.

The challenge in implementing a recoverable and detectable *FAA* operation is that some return values may be lost upon a crash, if they were not persisted. Such operations may have already affected the global counter, i.e., the return values of other operations. Upon recovery, it is necessary to figure out the return values of incomplete operations so that all operations (completed and pending) can be linearized. For example, in Figure 1, all operations, including pending ones, must be linearized after the system recovers from the

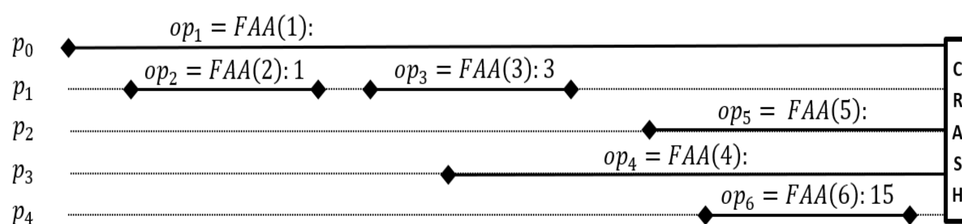

Figure 1 Challenges in detectable FAA implementation. We use the notation $op = FAA(V_1) : V_2$ to denote an FAA invocation with argument V_1 that returns V_2 as its response. An empty V_2 means that op is pending when the crash occurs.

crash. This is because all of them succeeded in adding their argument to the counter, and thus, have impacted other operations. Note that p_0's pending operation must be linearized right before the first operation of p_1 that is affected by it. On the other hand, we can choose where to linearize the pending operations of p_2 and p_3, as long as they are linearized before the operation of p_4, which is affected by both. We track and identify this information using a combination of sequence numbers and vector timestamps, as explained next.

Each FAA operation by a process has a strictly increasing sequence number; thus, it is uniquely identified by the pair ⟨process id, sequence number⟩. Linearizing FAA operations is facilitated with a global unbounded *fetch&add* base object. This object holds a *vector timestamp* with the sequence numbers of the last FAA operation of each process. Each FAA operation updates this object, and obtains a timestamp that precisely tracks the FAA operations affecting it. This tracking allows to ensure consistency with completed operations whose arguments were already added to the implemented counter. Since sequence numbers are unbounded, the *fetch&add* base object is unbounded as well.

The algorithm also uses two arrays: the first holds details of each FAA operation and the other helps to persist the operation and ensure consistency during recovery.

An FAA operation has three stages: it is first announced in the first array; then, its sequence number is updated at the *fetch&add* base object; finally, its vector timestamp, which can be used to compute its return value, is persisted in the second array.

Recovery is done by a single process, which is responsible for determining the return values of all operations that were pending when the crash occurred, depending on which stage they were at. If the operation did not modify the base *fetch&add* object, its return value indicates that it should be re-executed. If the operation wrote to the second array, it has been persisted and its return value is known. The main challenge is to determine the return value of an operation that modified the base *fetch&add* object but did not persist its return value in the second array. To handle these operations, the recovery process first orders the persisted operations, and then finds a place to insert each of these operations, so that its return value is consistent with the return values of the persisted operations.

3.1 Shared Data Structures

The algorithm maintains a global array $Seq[N]$ holding the sequence numbers of the current FAA operation of each process, all initially zero.

The $Res[N]$ array is used only in case of a crash, and it holds return values for all processes after the recovery ends; $\perp$ indicates that the FAA operation did not take effect and should be re-executed. All components in $Res[N]$ are initially $\perp$.

0	...	$N-1$	N	...	$2N-1$	...	kN	...	$(k+1)N-1$	...
1	0	0	1	0	1	0	0	0	0	0

Figure 2 The organization of W: Process 0's assigned bits, 0 and N, store $11_{(2)}$, while process $N-1$'s assigned bits, $N-1$ and $2N-1$, store $10_{(2)}$.

The *TotalContrib*$[N][\infty]$ array stores the intermediate sums of contributions of each process. *TotalContrib*$[i][k]$ holds the sum of the additions of all FAA operations executed by process i until and including the k-th operation. All components in *TotalContrib*$[i]$ are initially zero.

▶ **Definition 1.** *A* vector timestamp *(*VTS*) holds N sequence numbers, one for each process. $VTS_1 \prec VTS_2$ if VTS_2 is larger than or equal to VTS_1 in each component and $VTS_1 \neq VTS_2$. Two VTSs are* comparable *if one of them is larger than the other, in the $\prec$ order.*

Each FAA *op* by process i has an associated VTS_{op}, indicating the sequence numbers of the FAA operations that precede it: $VTS_{op}[j]$ is the sequence number of the last FAA operation by process j that precedes *op*.

An *OpVTS*$[N][\infty]$ array stores the persistence information of operations of each process. *OpVTS*$[i][k]$ holds the VTS associated with process i's k-th FAA operation; all components in *OpVTS*$[i]$ are initially $\perp$.

These data structures are accessed only with read and write primitives: process i reads and writes only the i-th component of each of them, while the recovery process reads and writes all the components.

The base *fetch&add* object. The algorithm uses an unbounded-size register W that is accessed by all processes with *fetch&add* primitives. W holds for each process i the sequence number of the last FAA operation process i executed. W's value is numeric and is changed with *fetch&add*. Since the sequence numbers stored in W are unbounded, they are stored and manipulated as follows (see Figure 2):

The sequence number of process i is stored in bits $k * N + i$, $k \in [0, \infty)$. To increment its sequence number in W, process i has to add a value that will set and clear corresponding bits, considering the previous stored value, so the new value is stored correctly in i's bits. For example, assume process 0's bits store $101_{(2)}$, i.e, bits 0 and $2N$ are set. After the increment, process 0's bits should store $110_{(2)}$ so bit$N$ should be set and bit $2N$ should be cleared, which is done by applying *fetch&add* with argument $2^N - 2^{2N}$.[2].

Process i uses two functions to manipulate W:

ReadVTS: applies $W.faa(0)$ in order to read W's content and returns a VTS of the N sequence numbers stored in it.

IncrementSeqAndGetVTS: increments process i's sequence number stored in W using a single primitive atomic *fetch&add*. The function returns a VTS out of the previous value of W returned by the *fetch&add*.

[2] Although the number of bits assigned in W to each process is unbounded, the number of bits that are actually used by process i can be determined according to the value of *Seq*[i]

3.2 Code Description

The pseudo code appears in Algorithm 1. Recall that $Seq[i]$ is incremented by 1, before the FAA operation is invoked. Process i starts an FAA operation by reading its prior total contribution, until the previous operation (Line 2). In Line 3, it declares the new operation by storing the new total contribution in $TotalContrib[i][Seq[i]]$; this value is the sum of *prevTotalContrib* and *val*, the argument of the current FAA operation by process i. In Line 4, process i's sequence number stored in W is incremented using *IncrementSeqAndGetVTS*, which returns a vector, VTS, of N sequence numbers. These sequence numbers indicate, for each process, the last FAA operation prior to process i's operation $\langle i, Seq[i]\rangle$. Line 5 stores VTS in $OpVTS[i][Seq[i]]$, thereby persisting the operation. Finally, the FAA operation returns $ComputeVal(VTS)$. As shown in Algorithm 1, *ComputeVal* with argument VTS sums $TotalContrib[p][VTS[p]]$, for all processes p, i.e., p's total contribution until and including the operation whose sequence number is stored in $VTS[p]$.

▶ **Definition 2.** *An operation is* invisible *if it did not perform the fetch&add primitive in Line 4. An invisible operation does not update its sequence number in W and does not affect other processes. An operation is* effective *if it performed the fetch&add primitive in Line* 4*, and updated its sequence number in W. An effective operation is* persisted *if it performed Line 5, so its* VTS *is persisted and its return value can be calculated based on it.*

Incrementing $Seq[i]$ before an FAA operation is invoked allows to distinguish, during recovery, between an invisible operation that was just invoked and a prior persisted operation.

The *RECOVER* function is executed by a single process. In Line 10, it collects all persisted operations in *persistedOps*. These are the operations whose VTSs appear in *OpVTS*. Then, it creates an order, L_0, of *persistedOps* according to the order of their VTSs (Line 11). Note that the VTSs of persisted operations are comparable. The main loop (Line 13) recovers the last operation of each process p, depending on its type; The sequence number of the operation, seq_p, is read from W, using *ReadVTS(W)[p]* (Line 14). If seq_p is smaller than $Seq[p]$ (Line 15), then process p did not execute the *fetch&add* in Line 4 before the crash. Thus, $op_p = \langle p, Seq[p]\rangle$ is invisible and should be re-executed. Otherwise (Line 17), if VTS is in $OpVTS[p][seq_p]$, then process p executed Line 5 and persisted its operation by storing the corresponding VTS in $OpVTS[p][seq_p]$. In this case, *ComputeVal* is applied to the corresponding VTS in order to compute the return value. We note that processes that did not invoke any FAA operation before the crash, with $Seq[p] == 0$, are skipped. The remaining case is when the operation is effective but non-persisted. In this case, *RECOVER* extends the order created in the previous iteration of the loop (initially, L_0) by inserting the operation into it. This is done with *InsertOperationIntoOrder*, explained next.

The function *InsertOperationIntoOrder* gets an operation $op_p = \langle p, seq_p\rangle$ to insert, L_0, the ordering of persisted operations,and L_{k-1}, the ordering after the previous effective non-persisted operation was inserted. The function finds the *barrier* of op_p, which is the smallest operation in L_0 that follows op_p, i.e., with $VTS_{barrier}[p] = seq_p$. The function creates L_k by placing op_p as the immediate predecessor of its barrier; if no such barrier operation exists, then op_p is placed at the end of L_{k-1} to create L_k. This ensures consistency with the persisted operations.

In Lines 25-30, the function derives the VTS corresponding to op_p from its immediate predecessor in L_k. If there is no immediate predecessor, then op_p is the first operation that applied *fetch&add* to W, and VTS_p is defined as all zeros (Line 26). Otherwise, let $op_t = \langle t, seq_t\rangle$ be the immediate predecessor. The function sets VTS_p to be op_t's corresponding vector time stamp, $OpVTS[t][seq_t]$, except that its t-th component is set to seq_t, indicating that op_p is the immediate successor of op_t. In Line 30, the function persists VTS_p in $OpVTS[p][seq_p]$, to safeguard against future crashes, and returns L_k.

Algorithm 1 Recoverable detectable FAA, for the global-crash model.

```
1:  procedure FAA(val)                                              ▷ executed by process i
2:      prevTotalContrib ← TotalContrib[i][Seq[i]-1]
3:      TotalContrib[i][Seq[i]] ← prevTotalContrib + val          ▷ Store current total contrib
4:      VTS ← IncrementSeqAndGetVTS(W, i)
5:      OpVTS[i][Seq[i]]← VTS                                               ▷ Store VTS
6:      return ComputeVal(VTS)                                     ▷ Compute return value
7:  procedure ComputeVal(VTS)
8:      return ∑_{process p} TotalContrib[p][VTS[p]]
9:  procedure RECOVER()                                      ▷ executed by recovery process
10:     persistedOps ← all operations whose VTSs appear in OpVTS
11:     L_0 ← persistedOps ordered according to their VTSs
12:     prevOrder ← L_0
13:     for p from 0 to N − 1 do
14:         seq_p ← ReadVTS(W)[p]
15:         if seq_p < Seq[p] then                                  ▷ invisible operation
16:             Res[p] = ⊥
17:         else if OpVTS[p][seq_p] ≠⊥ then                         ▷ persisted operation
18:             Res[p] = ComputeVal(OpVTS[p][seq_p])
19:         else                                   ▷ effective but non-persisted operation
20:             prevOrder ← InsertOperationIntoOrder(⟨p, seq_p⟩, L_0, prevOrder)
21:             Res[p] = ComputeVal(OpVTS[p][seq_p])
22: procedure InsertOperationIntoOrder(⟨p, seq_p⟩, L_0, L_{k−1})
23:     barrier ← smallest operation in L_0 such that VTS_{barrier}[p] = seq_p
24:     Insert ⟨p, seq_p⟩ as the immediate predecessor operation to barrier in L_{k−1} to get L_k.
        If there is no such barrier, append ⟨p, seq_p⟩ at the end of the order to get L_k.
25:     ⟨t, seq_t⟩ ← immediate predecessor operation to ⟨p, seq_p⟩ in L_k
26:     if ⟨t, seq_t⟩ =⊥ then VTS_p ← all N components are zeros
27:     else
28:         VTS_p = OpVTS[t][seq_t]
29:         VTS_p[t] = seq_t
30:     OpVTS[p][seq_p]← VTS_p
31:     return L_k
```

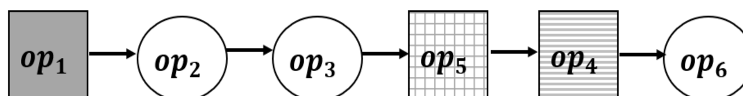

Figure 3 Linearization construction. L_0 orders op_2, op_3, op_6; L_1 is obtained by adding op_1; L_2 is obtained by adding op_5; L_3 is obtained by adding op_4.

Then, *RECOVER* applies *ComputeVal* to $OpVTS[p][seq_p]$, in order to compute the return value. Notice that there are at most N effective non-persisted operations, at most one for each process. The function recovers the processes from 0 to $N-1$. By Line 24, effective non-persisted operations with the same barrier are ordered in an ascending order of their processes' ids. Moreover, after the function finds the place for such operation, it builds its corresponding VTS based on the VTS of its immediate predecessor. This implies that the VTS of the immediate predecessor operation is already written when we use it.

In Lines 16, 18 and 21, the function saves the return values of each process in the *Res* array so when the processes resume, they can read their correct return values. If $Res[i] == \perp$ then the process should re-execute the FAA operation, i.e., proceed from Line 2. Otherwise, $Res[i]$ holds the correct return value for process i.

▶ **Example 3.** Consider the execution of Figure 1, ending with a crash. Assume all pending operations are effective and non-persisted. Figure 3 illustrates the order of the operations build by the *RECOVER* procedure, as follows. L_0 orders the persisted operations $\{op_2, op_3, op_6\}$ (empty circles). The effective non-persisted operations op_1, op_5, op_4 which are represented by squares, executed by processes p_0, p_2, p_3, respectively, are considered in the order of their processes' identifiers. The barrier of op_1 is op_2, so op_1 (filled square) is placed as op_2's immediate predecessor to obtain L_1. Next, the barrier of op_5 is op_6, so op_5 (squares pattern) is placed as op_6's immediate predecessor to obtain L_2. Finally, op_6 is also the barrier of op_4, so op_4 (stripes pattern) is placed as op_6's new immediate predecessor with op_5 as its immediate predecessor to obtain L_3. Note that operations with the same barrier could be linearized arbitrarily, and we chose to follow an ascending order of process' ids.

By Figure 1, the return values of op_2, op_3, op_6 are $1, 3, 15$, respectively. op_1's return value is 0, as it is the first operation applied to the *fetch&add* object. op_5's return value is 6, which is the sum of the return value of its predecessor op_3 and op_3's contribution, that is, the sum of the contributions of all op_5's preceding operations. op_4's return value is 11, which is the sum of the return value of its predecessor op_5 and op_5's contribution.

3.3 Correctness Proof

Let α be a crash-free execution of Algorithm 1. Each effective operation has an associated VTS. We order VTSs by coordinate-wise comparison as defined in Definition 1.

▶ **Lemma 4.** *The VTSs of two effective operations are comparable; furthermore, this order respects the order in which the corresponding operations executed the atomic fetch&add in Line 4, and hence, their real-time order.*

Proof. The lemma follows from the atomicity of the *fetch&add* performed in Line 4 and from the fact that each process executing Line 4 increments its sequence number stored in W by 1. Thus, every time Line 4 is executed, exactly one sequence number in W is changed and increased. That is, every time Line 4 is executed, the returned VTS is larger than its

immediate predecessor in exactly one component. Thus, comparing two VTSs, at least one component is larger than the other and all other components are larger or equal. Therefore they are *comparable*. Furthermore, the VTS of an operation op_2 that executes Line 4 after another operation op_1 is larger than the VTS of op_1 by the $\prec$ order. Consequently, $\prec$ respects the execution's real-time order. ◀

We linearize the effective operations in α according to their VTSs. Call this order L. Since all persisted operations are effective, they are linearized. Non-effective operations, i.e., invisible operations, are omitted and not linearized.

▶ **Lemma 5.** *The return values in L respect the sequential specification of Fetch&Add, i.e., they are the sum of the arguments of all preceding operations.*

Proof. Let op' be an effective operation and let $VTS_{op'}$ be its associated VTS. The proof is by induction on the position of op' in L. The base case is that op' is the smallest operation in the order, i.e, is the first FAA operation that executed Line 4 on W. Therefore, all components in $VTS_{op'}$, returned in Line 4, are zeros. For each i, $TotalContrib[i][0] = 0$ by the initialization of *TotalContrib*. Therefore, *ComputeVal* returns 0, which respects the sequential semantics of *Fetch&Add* because initially, the value of the implemented object is 0.

Assume the lemma holds for all operations smaller than op' in L. Assume operation op is the immediate predecessor operation to op' in L. Thus, by induction this implies that $res_{op} = ComputeVal(VTS_{op})$ respects the sequential semantics of *Fetch&Add*. The previous value of the implemented object, before operation op is performed, is res_{op}. Assume op is executed by process i, therefore by executing Line 4, i's sequence number is incremented by 1. Therefore, $VTS_{op'}[i]$ is larger than $VTS_{op}[i]$ by 1 and equals to the sequence number of op, seq_i. All other components are equal. For each process p, let $contrib_p$ and $contrib'_p$ be the total contributions of p, as functions $ComputeVal(VTS_{op})$ and $ComputeVal(VTS_{op'})$ considered in their calculation, respectively. For each $p \neq i$, $contrib_p = contrib'_p$. For i, $TotalContrib[i][seq_i] = TotalContrib[i][seq_i - 1] + val_{op}$, therefore, $contrib'_i = contrib_i + val_{op}$. Therefore, $ComputeVal(VTS_{op'})$ equals res_{op} plus the new value was added by op.

This respects the sequential semantics of *Fetch&Add* because the value of the implemented object before op' is applied to it is its value immediately before op is applied to it plus the value added by op. ◀

Let α' be an execution that ends with a global crash. The VTS of any persisted operation appears in $OpVTS$. Let L_0 be the sequence of all persisted operations, ordered according to their VTSs.

Next consider the last FAA operation of each process. If it is invisible, we assign the return value $\perp$. If it is persisted, we assign the return value computed by *ComputeVal* on its associated VTS. Finally, for an effective operation that is non-persisted, we find a place among the persisted operations and extend L_0. Since each process has at most one effective non-persisted operation, we can consider them by the order of their ids. Finding a place for the k-th effective non-persisted operation yields L_k which extends L_{k-1}.

Given L_{k-1}, let $op = \langle i, seq_i \rangle$ be the k-th effective non-persisted operation. The *barrier* of op is the smallest operation in L_0 that follows op, i.e., $VTS_{barrier}[i] = seq_i$. op is inserted right before its barrier in L_{k-1}, to get L_k. If there is no such operation, op is appended at the end of L_{k-1} to get L_k. Let L be the final linearization, after all effective non-persisted operations were inserted. A simple induction on k proves the next claim:

▷ Claim 6. Let op be the effective non-persisted operation that is inserted in L_k. Its immediate predecessor in L_k stays the same in L.

We compute the return values of effective non-persisted operation, based on the return value of the operation linearized immediately before them and its associated VTS, as follows: assume op' is an effective non-persisted operation and let op be its immediate predecessor executed by process i. The return value of op' is *ComputeVal*($VTS_{op'}$) while $VTS_{op'}$ equals VTS_{op} except for the i-th component in which $VTS_{op'}[i] = seq_i$, the sequence of op.

▷ Claim 7. Let op be a persisted operation ordered in L_0. The operations preceding op in L are consistent with its stored VTS_{op}. That is, an operation of some process p precedes op in L if and only if its sequence number is smaller than or equal to $VTS_{op}[p]$.

▶ **Lemma 8.** *The return values in L satisfy the sequential specification of Fetch&Add.*

Proof. The proof is by induction on the position of every effective operation op' in L; note that non-effective operations are not linearized in L.

Consider the operations in L; the base case is that op' is the first operation in L. If op' is persisted, the VTS associated with op' was stored in $OpVTS$ before the crash. By Claim 7, for each persisted op', the operations preceding op' in L are consistent with its stored $VTS_{op'}$. That is, the return value of op', which is $ComputeVal(VTS_{op'})$ is the sum of the arguments of all preceding operations in L.

Otherwise, op' is non-persisted without preceding operations. By Line 26, all components in $VTS_{op'}$ are zeros. For every i, $TotalContrib[i][0] = 0$ by the initialization and *ComputeVal* returns 0, respecting the sequential specification of *Fetch&Add*.

Assume the lemma holds for all operations that appear in L before op'. If op' is persisted, then the lemma holds as in the base case. Otherwise, let operation op executed by process t be the operation immediately preceding op' in L. By Claim 6, it is the same immediate predecessor that was used to build $VTS_{op'}$. By the assumption, $res_{op} = ComputeVal(VTS_{op})$ satisfies the lemma. The previous value of W before op was res_{op}. By the construction in Lines 28-29, VTS_{op} and $VTS_{op'}$ differ in the t-th component. $VTS_{op}[t] < VTS_{op'}[t] = seq_t$, the sequence number of op. Thus, $ComputeVal(VTS_{op'})$ takes for each process $k \neq t$, the same total contribution as $ComputeVal(VTS_{op})$ and for t, takes the total contribution of operation $\langle t, seq_t \rangle$. The latter is equal to the total contribution of operation $\langle t, seq_t - 1 \rangle$ plus val_{op}. That is, $ComputeVal(VTS_{op'})$ equals res_{op} plus the argument added by op. This respects the sequential specification because the value of W before op' should be the value before op plus the value added by op. ◀

▶ **Theorem 9.** *Algorithm 1 implements a recoverable detectable FAA in the global-crash model using only read, write and fetch&add primitive operations, and satisfies NRL.*

Complexity. In a crash-free execution, an *FAA* operation executes one *fetch&add* operation, a constant number of writes and $O(N)$ reads from shared memory during the *ComputeVal* function. The algorithm can be modified to store the total contribution values in W, using a similar encoding scheme. This would allow to read all contributions using a single shared memory access, yielding an FAA implementation with $O(1)$ crash-free complexity.

4 FAA Implementation in the Individual-Crash Model

In the individual-crash model, each process can crash individually without affecting executions of other processes, and then it also recovers individually. Thus, a process may crash and recover without the other processes being aware of this.

In addition to the data structures of Algorithm 1, we use the following data structures: $isInRecovery[N]$ holds in the i-th entry the sequence number of the last operation during whose execution process i crashed; all entries are initially 0. We also use $mutex$, an RME lock implemented using only read and write primitives [18]. RME ensures that if process i crashes while holding $mutex$, no other process can acquire the lock until i tries to acquire it again; in this case i will succeed, i.e., $mutex$ will still be held by i.

The pseudo code appears in Algorithm 2. FAA(val) is identical to the one for global crashes (Algorithm 1) and is omitted. Process i manipulates W and the other data structures using the same functions as in Section 3.1.

Algorithm 2 Recoverable Detectable FAA, for the individual-crash model.

```
32: procedure FAA.RECOVER(val)                                  ▷ executed by process i
33:     seq_i ← ReadVTS(W)[i]
34:     if seq_i < Seq[i] then                                  ▷ invisible operation
35:         re-execute FAA(val)
36:     else                                                    ▷ effective operation
37:         isInRecovery[i] ← seq_i
38:         await(mutex.lock())                                 ▷ lock robust to failures
39:         if OpVTS[i][seq_i] ≠⊥ then                          ▷ persisted operation
40:             mutex.release()
41:             return ComputeVal(OpVTS[i][seq_i])              ▷ from Algorithm 1
42:         recoveryW ← ReadVTS(W)
43:         for k from 0 to N-1 do  await(OpVTS[k][recoveryW[k]] ≠⊥ or
                        isInRecovery[k] == recoveryW[k])
44:         Ops ← all operations until sequence numbers in recoveryW.
45:         persistedOps ← all operations in Ops, whose VTSs appear in OpVTS
46:         L_0 ← order persistedOps according to their VTSs
47:         InsertOperationIntoOrder(⟨i, seq_i⟩, L_0, L_0)      ▷ from Algorithm 1
48:         mutex.release()
49:         return ComputeVal(OpVTS[i][seq_i])                  ▷ from Algorithm 1
```

FAA.RECOVER(val) is executed by each crashed process independently and includes acquiring a lock. Thus, the algorithm must consider repeated crashes during FAA.RECOVER of the same process such that a process that acquires the lock, crashes and then invokes FAA.RECOVER again (possibly several times), will eventually release the lock at the end of its recovery. Lines 33-35 have the same logic as Lines 14-16 in *RECOVER()* (Algorithm 1). In Line 33, the function reads the sequence number of the last operation of process i, seq_i, from W using *ReadVTS(W)[i]*. In Line 34, the function checks if seq_i is smaller than $Seq[i]$. If it is, process i did not execute the critical *fetch&add* inside *IncrementSeqAndGetVTS* before crashing, implying $op_i = \langle i, seq_i \rangle$ is invisible and should be re-executed. Otherwise, op_i is effective and can be persisted or non-persisted. In Line 37, the function declares that $\langle i, seq_i \rangle$ is in recovery by writing the sequence number of the current operation, seq_i, in $isInRecovery[i]$. Then, the process repeatedly attempts to acquire the $mutex$ lock in Line 38 until it succeeds (if it ever does).

Once process i acquires $mutex$ it proceeds to recover its operation op_i. First, it checks if op_i is already persisted (Line 39). This may occur in two scenarios: either op_i is a *pure* persisted operation, that is, i crashed only after updating its VTS in $OpVTS$; or, op_i was an effective non-persisted operation but i has already recovered and persisted its VTS in

$OpVTS$, but it crashed before returning, possibly while holding the lock. In both cases, process i releases the lock and returns a response based on its VTS. Otherwise, the VTS of op_i is not persisted in $OpVTS$. The function reads W using $ReadVTS$ and stores the returned VTS in a local variable $recoveryW[N]$ consisting of N sequence numbers (Line 42). $recoveryW$ represents a value of W in a specific point in time. Process i treats $recoveryW$ similarly to how W is treated by the recovery code for the global-crash model and will order op_i according to it. In Line 43, process i waits until each process k persists the operation with the sequence stored in $recoveryW[k]$ or is in its recovery code on that operation. We note that processes that did not invoke an FAA operation before the await condition, with $Seq[p] == 0$, are skipped.

After the loop of Line 43 terminates, all operations in $recoveryW$ are either persisted or effective and non-persisted but in recovery. In Line 44, the function stores in the set *Ops*, for each process, all its operations whose sequence number is smaller or equal to that stored for the process in $recoveryW$. We then proceed in a way similar to the global-crash algorithm. Process i collects into *persistedOps* all persisted operations in *Ops*, i.e., whose VTSs appear in $OpVTS$ (Line 45), and creates the order L_0 on *persistedOps* based on their VTSs (Line 46). Then, process i linearizes op_i using *InsertOperationIntoOrder* while the $prevOrder$ parameter is also L_0 because unlike the global-crash model, here, we insert a single operation, op_i. Note that each time a process acquires the lock and recovers its effective non-persisted operation, *op*, it persists it in $OpVTS$, such that the next process will consider *op* as a persisted operation and will order it as part of L_0. Therefore, effective non-persisted operations that have the same *barrier* are ordered by the real-time order of processes capturing the lock. Finally, i orders op_i, releases *mutex* (Line 48), and applies *ComputeVal* to the corresponding VTS of op_i to compute its response.

The correctness proof of this algorithm appears in Appendix A.1.

5 Impossibility of Wait-Free Recovery in the Individual-Crash Model

▶ **Theorem 10.** *There is no detectable self-implementation of FAA in the individual-crash model, such that both the FAA operation and the FAA.RECOVER function are lock-free.*

Proof. We prove the theorem using valency arguments [2, 12]. Assume, by way of contradiction, that there is such an implementation with lock-free FAA and $FAA.RECOVER$. In all executions we consider, the initial value of the FAA object is 0 and both processes, p and q, invoke FAA with argument 1. Hence, one process must return 0 and the other must return 1.

Given a configuration C and a process $r \in \{p, q\}$, we say that C is *r-valent* if there is a crash-free execution starting from C in which the return value of FAA or $FAA.RECOVER$ by r is 0. C is *bivalent* if it is both p-valent and q-valent. C is *p-univalent* if it is p-valent and not q-valent, and symmetrically for *q-univalent*. We say that C is *univalent* if it is either p-univalent or q-univalent.

The initial configuration, C_0, is bivalent because a solo execution of each process returns 0. Following a standard valency argument and since we assume that the FAA operation is lock-free, there is an execution starting from C_0 that leads to a bivalent configuration C_1, in which both p and q are about to take a *critical* step, i.e, a step that leads to a univalent configuration, one of which is p-univalent while the other is q-univalent. A standard argument can be used to show the following claim (see Appendix A.2):

▷ Claim 11. The critical steps of p and q apply *fetch&add* to the same base object.

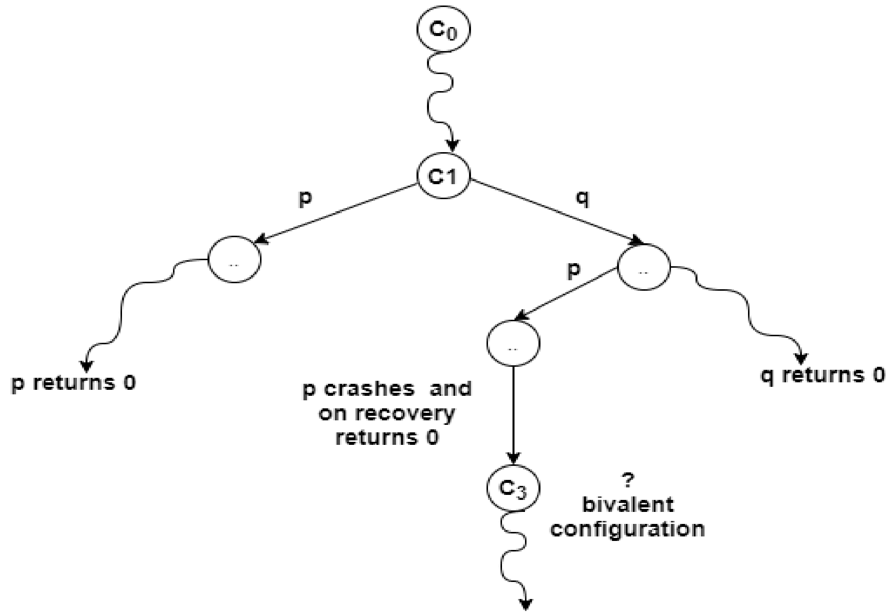

Figure 4 Illustration of one case in the proof of Theorem 10.

Assume, without loss of generality, that configuration $C_1 \circ p$ is p-univalent while $C_1 \circ q$ is q-univalent. Consider the execution from C_1 in which p takes a step followed by a step of q and then p crashes: $C_2 = C_1 \circ p \circ q \circ CRASH_p$.
Consider also the execution from C_1 in which q takes a step followed by a step of p and then p crashes: $C_3 = C_1 \circ q \circ p \circ CRASH_p$.

A solo execution of $FAA.RECOVER$ by p from both configurations C_2 and C_3 must complete, since it is lock-free. Furthermore, these two configurations are indistinguishable to p, because p's response from the primitive *fetch&add* is lost, while the value of the *fetch&add* base object is the same in both configurations. Therefore, an execution of $FAA.RECOVER$ by p from both C_2 and C_3 returns the same value – v.

Assume v is 0, and thus C_3 is p-valent. The configuration $C_1 \circ q \circ p$ is q-univalent, while $C_3 = C_1 \circ q \circ p \circ CRASH_p$ is p-valent. However, these configurations are indistinguishable to q because it is unaware of p's crash, therefore a solo execution of q from C_3 must return 0, that is, C_3 is q-valent. This proves that C_3 is bivalent (see Figure 4). The case $v = 1$ is symmetric, since if p returns 1 this proves the configuration is q-valent, as a solo execution of q after p completes must return 0; a similar argument proves that C_2 is bivalent.

In this manner, we can keep extending the execution to obtain a crash-free execution of q in which it performs an infinite number of steps without completing a single FAA operation, contradicting the assumption that the algorithm is lock-free. ◀

6 Discussion

We present two self-implementations of a recoverable detectable FAA operation, one for the global-crash model and the other for the individual-crash model. Both algorithms are wait-free in crash-free executions. Recovery in both algorithms is blocking. In the global-crash model, this is the result of a design choice to delegate recovery to a single process. For the individual-crash model, we prove that a lock-free self-implementation of a detectable FAA does not exist.

The proof of Theorem 10 constructs an execution in which one process, repeatedly crashing during its recovery, blocks another process that does not crash, from making progress. This leaves open the question of making progress once no process crashes. We note that Algorithm 2 may have an execution, where process i acquires the lock during its recovery and then crashes; this means that another process j that crashes and tries to recover, cannot complete its recovery, even when no further crashes occur. Finding an algorithm that makes progress when processes stop crashing, or proving such an algorithm does not exist, is an interesting question.

Our algorithms apply *fetch&add* primitives to a shared unbounded base object storing a vector timestamp with N entries. It would be interesting to see if the amount of memory storage can be bounded. This might be challenging, since the vector timestamp is used to precisely track which operations affected each persisted *FAA*, and detect where they should be linearized and with which return value. Another interesting open question is whether there exists a self-implementation of a recoverable *FAA* object for the global-crash model such that both the *FAA* operation and the recovery code are wait-free.

References

1. M. K. Aguilera and S. Frølund. Strict linearizability and the power of aborting. HPL-2003-241.
2. Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *PODC*, pages 7–16, 2018.
3. Ohad Ben-Baruch, Danny Hendler, and Matan Rusanovsky. Upper and lower bounds on the space complexity of detectable objects. In *PODC*, 2020.
4. Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *SPAA*, pages 253–264, 2019.
5. Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *OPODIS*, pages 20:1–20:17, 2016.
6. Dhruva R. Chakrabarti, Hans-Juergen Boehm, and Kumud Bhandari. Atlas: leveraging locks for non-volatile memory consistency. In *OOPSLA*, pages 433–452, 2014.
7. D. Y. C. Chan and P. Woelfel. Recoverable mutual exclusion with constant amortized RMR complexity from standard primitives. In *PODC*, 2020.
8. Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The inherent cost of remembering consistently. In *SPAA*, pages 259–269, 2018.
9. Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *SPAA*, pages 271–282, 2018.
10. Andreia Correia, Pascal Felber, and Pedro Ramalhete. Persistent memory and the rise of universal constructions. In *EuroSys*, pages 5:1–5:15, 2020.
11. Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *USENIX*, pages 373–386, 2018.
12. Michael J. Fischer, Nancy A. Lynch, and Michael Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
13. Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. NVTraverse: In NVRAM data structures, the destination is more important than the journey. In *PLDI*, pages 377–392, 2020.
14. Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. Brief announcement: A persistent lock-free queue for non-volatile memory. In *DISC*, pages 50:1–50:4, 2017.
15. Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *PPoPP*, pages 28–40, 2018.
16. Wojciech Golab. The recoverable consensus hierarchy. In *SPAA*, pages 281–291, 2020.
17. Wojciech Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *PODC*, pages 211–220, 2017.

18 Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. In *PODC*, pages 65–74, 2016.

19 Wojciech M. Golab and Danny Hendler. Recoverable mutual exclusion under system-wide failures. In *PODC*, pages 17–26, 2018.

20 R. Guerraoui and R. R. Levy. Robust emulations of shared memory in a crash recovery model. In *ICDCS*, pages 400–407, 2004.

21 Maurice Herlihy. Wait free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, 1991.

22 Maurice P. Herlihy and Jennette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

23 J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *DISC*, pages 313–327, 2016.

24 Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *ASPLOS*, pages 427–442, 2016.

25 Prasad Jayanti, Siddhartha V. Jayanti, and Anup Joshi. Recoverable mutex algorithm with sub-logarithmic RMR on both CC and DSM. In *PODC*, pages 177–186, 2019.

26 Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dalí: A periodically persistent hash map. In *DISC*, pages 37:1–37:16, 2017.

27 Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. Onefile: A wait-free persistent transactional memory. In *DSN*, pages 151–163, 2019.

28 David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. Nvc-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *VLDB Workshop on In-Memory Data Mangement and Analytics*, pages 4:1–4:8, 2015.

29 Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. In *OOPSLA*, pages 128:1–128:26, 2019.

A Additional Proofs

A.1 Sketch of Correctness Proof for Individual-Crash Model

Correctness for crash-free executions is the same as in Section 3.3. We now consider an execution α with individual crashes.

▶ **Observation 12.** *Let α be an execution where i crashes during operation op_i. If op_i is effective non-persisted, its VTS is recovered and stored in OpVTS exactly once.*

▶ **Lemma 13.** *The return values of all operations executing Algorithm 2 satisfy the sequential specification of Fetch&Add.*

Proof. Note that in the individual-crash model, we do not refer to a final linearization order L, as in the global-crash model because each process only recovers its own operation by inserting it to an L_0 order of the persisted operations it currently read. Consider the L_0 order by process i. We say that a return value of each persisted operation, *op*, satisfies the sequential specification of *Fetch&Add* although not all effective non-persisted operations that *op* follows necessarily appear in L_0.

The proof relies on the following argument. Assume process i orders its effective non-persisted operation op_i based on the L_0 it computes, and let op_j be the immediate predecessor of op_i in the new order. Then, i sets VTS_{op_i} to be identical to VTS_{op_j} except for the j-th component where it is larger by 1. Therefore, the return value it computes is $res_{op_i} = res_{op_j} + val_{op_j}$. For any other non-persisted operation op_k one of the following holds. Either

op_k has been observed by VTS_{op_j}, that is, VTS_{op_j} in its k-th component contains a sequence number larger or equal to the sequence number of op_k. In such case, the response of op_j took into consideration the value added by op_k, and thus also the response of op_i. Moreover, we are guaranteed that if process k acquires the lock it will order op_k before op_i, since it orders it before the first VTS that observed it. Otherwise, op_k was not observed by VTS_{op_j}, thus not observed also by VTS_{op_i}, and both return values do not consider the value added by op_k. However, once k acquires the lock and orders op_k it will order it after op_i, since none of the preceding operations observed op_k.

This proves that res_{op_i} satisfies the sequential specification of *Fetch&Add*, since its return value considers all operations that precede it, and those operations will be ordered before op_i (if they are not persisted yet), and no other operation will be ordered before op_i. ◀

A.2 Proof of Claim 11

We consider all possible steps: read, write and $fetch\&add$. Assume s_p and s_q are critical steps by process p and q, receptively, such that $C_1 \circ s_p$ is p-univalent while $C_1 \circ s_q$ is q-univalent.

- Steps s_p and s_q access distinct registers. In this case, these configurations are indistinguishable to p and q, that is, $C \circ s_p \circ s_q \overset{p,q}{\sim} C \circ s_q \circ s_p$.
- Steps s_p and s_q read the same register. Also in this case, $C \circ s_p \circ s_q \overset{p,q}{\sim} C \circ s_q \circ s_p$.
- Step s_p writes to some register r step and s_q reads r. In this case, $C \circ s_p \overset{p}{\sim} C \circ s_q \circ s_p$ holds.
- Step s_p applies *fetch&add* and step s_q reads r. In this case, $C \circ s_p \overset{p}{\sim} C \circ s_q \circ s_p$ holds.
- Steps s_p and s_q write to the same register. In this case, $C \circ s_p \overset{p}{\sim} C \circ s_q \circ s_p$ holds.
- Step s_p applies *fetch&add*, step s_q writes to the same register. In this case, $C \circ s_q \overset{q}{\sim} C \circ s_p \circ s_q$ holds.
- Step s_p applies *fetch&add* with $val = 0$, step s_q applies *fetch&add* to the same register. In this case, $C \circ s_q \overset{q}{\sim} C \circ s_p \circ s_q$ holds.

In each of the above cases, the configurations are indistinguishable to at least one process, and therefore, must have the same valencies. Therefore, it must be that p and q apply *fetch&add* to the same base object.