Durable Algorithms for Writable LL/SC and CAS with Dynamic Joining

Prasad Jayanti
Dartmouth College, USA

Siddhartha Jayanti
Google Research, USA

Sucharita Jayanti
Brown University, USA

Abstract

We present durable implementations for two well known universal primitives—CAS (compare-and-swap), and its ABA-free counter-part LLSC (load-linked, store-conditional). Our implementations satisfy *method-based recoverable linearizability (MRL)* and *method-based detectability (M-detectability)*—novel correctness conditions that require only a simple usage pattern to guarantee resilience to individual process crashes (and system-wide crashes), including in implementations with nesting. Additionally, our implementations are: *writable*, meaning they support a Write() operation; have *constant time complexity* per operation; allow for *dynamic joining*, meaning newly created processes (a.k.a. threads) of arbitrary names can join a protocol and access our implementations; and have *adaptive space complexity*, meaning the space use scales in the number of processes n that actually use the objects, as opposed to previous protocols whose space complexity depends on N, the maximum number of processes that the protocol is designed for. Our durable Writable-CAS implementation, DuraCAS, requires $O(m + n)$ space to support m objects that get accessed by n processes, improving on the state-of-the-art $O(m + N^2)$. By definition, LLSC objects must store "contexts" in addition to object values. Our Writable-LLSC implementation, DuraLL, requires $O(m+n+C)$ space, where C is the number of "contexts" stored across all the objects. While LLSC has an advantage over CAS due to being ABA-free, the object definition seems to require additional space usage. To address this trade-off, we define an *External Context (EC)* variant of LLSC. Our EC Writable-LLSC implementation is ABA-free and has a space complexity of just $O(m + n)$.

To our knowledge, our algorithms are the first durable CAS algorithms that allow for dynamic joining, and are the first to exhibit adaptive space complexity. To our knowledge, we are the first to implement any type of *durable* LLSC objects.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks; Theory of computation → Concurrency; Theory of computation → Concurrent algorithms; Theory of computation → Parallel algorithms; Theory of computation → Data structures design and analysis; Theory of computation → Distributed algorithms

Keywords and phrases durable, recoverable, detectable, persistent memory, dynamic joining, LL/SC, CAS

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.25

Related Version *Full Version*: `https://arxiv.org/abs/2302.00135` [30, 34]

Funding Prasad Jayanti was supported in part by the James Frank Family Professorship. Siddhartha Jayanti was supported in part by an NDSEG Graduate Research Fellowship.

1 Introduction

The advent of *Non-Volatile Memory (NVM)* [27] spurred the development of durable algorithms for the *crash-restart model.* In this model, when a process π crashes, the contents of memory *persist* (i.e., remain unchanged), but π's CPU registers, including its program counter, lose their contents. To understand the difficulty that arises from losing register contents, suppose that π crashes at the point of executing a hardware CAS instruction, $r \leftarrow \textsc{Cas}(X, \mathit{old}, \mathit{new})$, on a memory word X and receiving the response into its CPU register r. When π subsequently restarts, π cannot tell whether the crash occurred before or after the CAS executed, and if the crash occurred after the CAS, π cannot tell whether the CAS was successful or not. Researchers identified this issue and proposed software-implemented *durable objects* [28, 5], which allow a restarted process to *recover* from its crash and *detect* the result of its last operation. The rapid commercial viability of byte-addressable, dense, fast, and cheap NVM chips has made efficient durable object design important.

Writable and non-Writable CAS. The Compare-and-Swap (CAS) instruction is ubiquitous in multiprocessor computation, both in concurrent and parallel algorithms. Recently, there has been a lot of research on implementing durable CAS objects because they are widely employed in practice and are universal: any durable object can be implemented from durable CAS objects [7, 25, 28]. Formally, the state of a CAS object X is simply its value, and the operation semantics are as follows:

- $X.\textsc{Cas}(\mathit{old}, \mathit{new})$: if $X = \mathit{old}$, sets X to *new* and returns *true*; otherwise, returns *false*.
- $X.\textsc{Read}()$: returns the value of X.
- $X.\textsc{Write}(\mathit{new})$: sets X to *new* and returns *true*.

If the object supports all three operations, it is a *Writable-CAS (W-CAS)*; and if it does not support $\textsc{Write}()$, it is a *non-Writable-CAS (nW-CAS)* object.

CAS's ABA problem and LLSC. Although CAS objects are powerful tools in concurrent computing, they also have a significant drawback called the *ABA-problem* [16]. Namely, if a process π reads a value A in X and executes $X.\textsc{Cas}(A, C)$ at a later time, this CAS will succeed *even if* the value of X changed between π's operations, from A to B and then back to A. So while any object *can* be implemented from CAS, the actual process of designing an algorithm to do so becomes difficult. In the non-durable setting, the ABA-problem is often overcome by using the hardware's double-width CAS primitive—in fact, "CAS2 [double-width CAS] operation is the most commonly cited approach for ABA prevention in the literature" [16]. However, all known durable CAS objects, including ours, are only one-word wide—even as they use hardware double-width CAS [5, 7, 6]. Against this backdrop, the durable LLSC objects presented in this paper serve as an invaluable alternate tool for ABA prevention.

LLSC objects are alternatives to CAS objects that have been invaluable in practice, since they are universal and ABA-free [38]. The state of an LLSC object Y is a pair $(Y.\mathit{val}, Y.\mathit{context})$, where $Y.\mathit{val}$ is the *value* and $Y.\mathit{context}$ is a set of processes (initially empty). Process π's operations on the object have the following semantics:

- $Y.\text{LL}()$: adds π to $Y.\mathit{context}$ and returns $Y.\mathit{val}$.
- $Y.\text{VL}()$: returns whether $\pi \in Y.\mathit{context}$.
- $Y.\text{SC}(\mathit{new})$: if $\pi \in Y.\mathit{context}$, sets Y's value to *new*, resets $Y.\mathit{context}$ to the empty set and returns *true*; otherwise, returns *false*.
- $Y.\textsc{Write}(\mathit{new})$ changes Y's value to *new* and resets $Y.\mathit{context}$ to the empty set.

The object is *Writable (W-LLSC)* or *non-Writable (nW-LLSC)* depending on whether the $\textsc{Write}()$ operation is supported.

To our knowledge, there are no earlier durable implementations of ABA-free CAS-like objects, including LLSC.

Wider impact of durable primitives. Durable primitives such as W-CAS and W-LLSC are particular important since they facilitate a plethora of other durable data structures. In particular, let A be an algorithm for implementing a data structure DS using either the read, write, CAS, or the read, write, LL/SC/VL instructions. Then, using durable W-CAS and durable W-LLSC, we can design an algorithm A' that implements a *durable* version of DS from hardware read, write, and CAS, without affecting the asymptotic time or space complexity [5, 7].

Previous work and the state-of-the-art. CAS and LLSC objects share close ties, but they also pose different implementational challenges. In the non-durable context, it is well known that non-writable LLSC (nW-LLSC) objects can be implemented from nW-CAS objects and visa versa in constant time and space. The simple implementation of nW-LLSC from nW-CAS however, requires packing a value-context pair into a single nW-CAS object [3]. Solutions that implement a full-word nW-LLSC from a full-word nW-CAS require a blow-up in time complexity, space complexity, or both [37, 18, 40, 38, 11]. Writability complicates the relationship further. Even in the non-durable context, reductions between W-CAS and W-LLSC have resulted in a blow-up in space complexity and fixing the number of processes *a priori* [29]. Writability can sometimes be added to an object that is non-writable, but this leads to an increase in space complexity [1].

There are no previous works on Durable LLSC. Three previous works have implemented durable CAS objects, all from the hardware CAS instruction: Attiya, Ben-Baruch, and Hendler [5], Ben-Baruch, Hendler, and Rusanovsky [6], and Ben-David, Blelloch, Friedman, and Wei [7]. All three papers provide implementations for a fixed set of N processes with *pid*s $1, \ldots, N$, and achieve constant time complexity per operation. Attiya et al. pioneered this line of research with a durable nW-CAS implementation, which achieves constant time complexity and requires $O(N^2)$ space per object. Ben-Baruch et al. present an nW-CAS implementation with optimal bit complexity. Their algorithm however, requires packing N bits and the object's value into a single hardware variable. Thus, if the value takes 64 bits, then only 64 pre-declared processes can access this object. (Current commodity multiprocessors range up to 224 cores [24], and can support orders-of-magnitude more threads.) Ben-David et al. designed an algorithm for nW-CAS, and then leveraged Aghazadeh, Golab, and Woelfel's writability transformation [1] to enhance that algorithm to include a Write operation, thereby presenting the only previous Writable-CAS implementation. Their nW-CAS algorithm uses a pre-allocated help-array of length $O(N)$, and their W-CAS algorithm uses an additional hazard-pointer array of length $O(N^2)$. Both arrays can be shared across objects, thus the implementation space complexities for m objects are $O(m+N)$ and $O(m+N^2)$, respectively.

Our contributions. We present four wait-free, durable implementations: DuraCAS for Writable-CAS, DuraLL for Writable-LLSC, DurEC for External Context (EC) nW-LLSC, and DurECW for EC W-LLSC (we will specify External Context LL/SC soon). Our implementations achieve the following properties:

1. Constant time complexity: All operations including recovery and detection run in $O(1)$ steps.
2. Dynamic Joining: Dynamically created processes of arbitrary names can use our objects.
3. Full-word size: Our implementations support full-word (i.e., 64-bit) values.
4. Adaptive Space Complexity: We quantify space complexity by the number of memory words needed to support m objects for a total of n processes. The DuraCAS, DurEC, and DurECW implementations require just constant memory per process and per object,

and thus each have a space complexity of $O(m+n)$. Since DuraLL must remember contexts, its space complexity is $O(m+n+C)$, where C is the number of contexts that must be remembered.[1]

We believe that our definitions and implementations of the External Context LLSC objects—which are ABA-free, space-efficient alternatives to CAS and LLSC—are of independent interest in the design of both durable and non-durable concurrent algorithms.

To our knowledge, our algorithms are the first durable CAS algorithms that allow for dynamic joining, and are the first to exhibit adaptive space complexity. To our knowledge, we are the first to consider any type of *durable* LLSC objects.

Our approach. We implement universal primitives that allow dynamic joining of new processes, have an *adaptive space complexity* that is constant per object and per process, and give an ABA-free option, while simultaneously achieving constant time complexity. Just like our predecessors, all our implementations rely on just the hardware double-width CAS instruction for synchronization.

A keystone of our approach is the observation that durable nW-LLSC—due to its ABA-freedom—serves as a better stepping stone than even durable nW-CAS on the path from hardware CAS to durable W-CAS. Perhaps less surprisingly, durable nW-LLSC is a great stepping stone towards durable W-LLSC also. However, by definition LLSC objects require more space to remember context for each process—an inherent burden that CAS objects do not have. Thus, using nW-LLSC objects in the construction of our W-CAS would lead to a bloated space complexity. To avoid this drawback, we define an *External Context (EC)* variant of LLSC. An EC LLSC object is like an LLSC object, except that its context is returned to the process instead of being maintained by the object. Thus, our EC nW-LLSC implementation, DurEC, is the building block of all our other implementations.

The state of an EC LLSC object Y is a pair $(Y.val, Y.seq)$, where the latter is a sequence number context. Process π's operations on the object have the following semantics:

- Y.ECLL(): returns $(Y.val, Y.seq)$.
- Y.ECVL(s): returns whether $Y.seq = s$.
- Y.ECSC(s, new): if $Y.seq = s$, sets Y's value to new, increases $Y.seq$, and returns *true*; otherwise, returns *false*.
- Y.Write(new): changes Y's value to new and increases $Y.seq$.

The object is *Writable (EC W-LLSC)* or *non-Writable (EC nW-LLSC)* depending on whether the Write() operation is supported.

We design durable implementations of External Context W-LLSC and W-CAS, called DurECW and DuraCAS, respectively; each implementation uses two DurEC base objects. We implement our durable W-LLSC algorithm, DuraLL, by simply internalizing the external contexts of a DurECW. All our implementations overcome the need for hazard-pointers and pre-allocated arrays for helping in order to allow dynamic joining and achieve adaptive space complexity. Key to eliminating these arrays are pointer based identity structures called *handles*, which we describe in Section 4. Figure 1 illustrates the differences between our approach and Ben-David et al.'s.

[1] C is the number of process-object pairs $(\pi, \mathcal{O})$, where π has performed an LL() operation on $\mathcal{O}$, and its last operation on $\mathcal{O}$ is not an SC() or Write(). A trivial upper bound is $C \leq nm$.

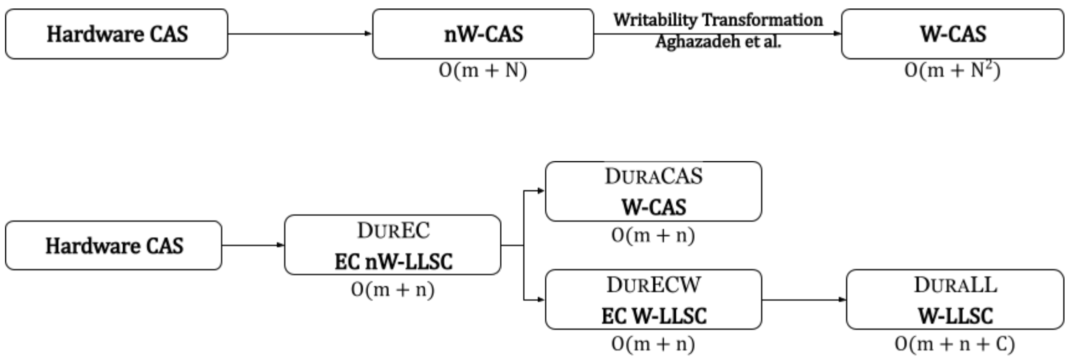

Figure 1 A comparison of Ben-David et al.'s approach (top) and our approach (bottom): each box represents an implementation—the type of the implementation is in bold and its space complexity appears below the box. The names of our implementations appear in the box in SMALLCAPS. An arrow from A to B means that B is implemented using A.

2 Related Work

We have already detailed the three previous works on durable CAS objects [5, 6, 7] in the introduction. In addition to these durable (single-word) CAS objects, there are implementations of durable multi-word CAS objects[2] by Wang, Levandoski, and Larson [42], and by Guerraoui, Kogan, Marathe, and Zablotchi [22]. Furthermore, LLSC can be implemented using multi-word CAS. However, these software implementations of multi-word CAS are lock-free but not wait-free, and they do not support the Write operation. Thus, using these algorithms, one can implement non-writable lock-free LL/SC, but not the writable and wait-free LL/SC primitive that our algorithm implements. Additionally, they require complex memory management which also leads to an increase in space complexity. We discuss other related work below.

Byte-addressable non-volatile memory laid the foundation for durable objects [27]. Research on durable objects has spanned locks [21, 41, 35, 36, 33, 31, 20, 12, 14, 17, 15, 32], and non-blocking objects—including queues [19], counters [5], registers [5, 6], CAS objects [5, 6, 7], and general transformations and universal constructions [28, 7, 4].

Several models of persistent memory systems have been proposed in the literature. In the *individual-crash model* [2, 5, 7, 10], processes can crash independently, while in the *system-crash model*, all processes crash together [28, 9, 19]. Izraelevitz et al. assume that crashed processes do not restart and the system spawns new processes with process ids that were never used before [28], but most other works assume processes restart with the same ids as before. Some works, such as [28, 19], model volatile caches: when a process performs a hardware operation on a shared variable, only the cached copy of the variable is updated, and the update transfers to the NVM only when the cache is explicitly flushed. Others work in the model that each step directly updates the variable's state in the NVM [5, 7, 21].

Several correctness criteria have been proposed for implementations: *recoverable linearizability* by Berryhill, Golab, and Tripunitara [9], *durable and buffered durable linearizability* by Israelevitz, Mendes, and Scott [28], *nested recoverable linearizability* by Attiya, Ben-Baruch,

[2] A k-word CAS, $\text{CAS}((X_1, \ldots, X_k), (old_1, \ldots, old_k), (new_1, \ldots, new_k))$, has the semantics: if all of $X_1 = old_1, \ldots, X_k = old_k$ then set $X_1 \leftarrow new_1, \ldots, X_k \leftarrow new_k$ and return *true*; otherwise, simply return *false*.

and Hendler [5], *persistent atomicity* by Guerraoui and Levy [23], and *strict linearizability* by Aguilera and Frølund [2]. These consistency criteria are surveyed by Ben-David, Friedman, and Wei [8].

Friedman, Herlihy, Marathe, and Petrank [19] identify that it is not enough for implementations to satisfy a linearizability condition, but they must also support *detection*, i.e., make possible for a process to find out whether its crashed operation took effect and, if it did, what the response was. Li and Golab [39] present a formulation of detectability.

3 Model

Our model is akin to those used in previous works on durable CAS [5, 7]. The system consists of asynchronous processes that communicate by applying atomic operations (Read or CAS) directly to shared variables stored in Non-Volatile Memory (NVM). We use the individual crash model where any process may crash at any time and restart at any later time, and the same process may crash and restart any number of times. When a process crashes, its registers, including its program counter, lose their contents (i.e., they are set to arbitrary values), but the contents of the NVM are unaffected. After a crash, a process eventually restarts (with the same process id as before).

To ensure that our objects are recoverable and detectable, we introduce two new correctness conditions for objects, called *Method-based Recoverable Linearizability (MRL)* and *Method-based Detectability (M-Detectability)*. MRL adapts and combines ideas from the well known notions of Recoverable Linearizability [9] and Nested-safe Recoverable Linearizability [5], and M-Detectability captures the corresponding notion of detectability [19].

MRL is "method-based" in the sense that it facilitates recoverability by requiring that an object $\mathcal{O}$ provide a method $\mathcal{O}.\textsc{Recover}()$ in addition to providing a method for each operation supported by $\mathcal{O}$. When there are no crashes, MRL reduces to standard linearizability [26]. In particular, if a process π invokes a method for an operation and completes the method without crashing, the operation is required to take effect atomically at some instant between the method's invocation and completion. On the other hand, if crashes occur, MRL guarantees the object remains consistent if the following *usage pattern* if followed. If π crashes after invoking some operation $\mathcal{O}.op$ and before that operation completes, when π subsequently restarts, the usage pattern requires that π execute $\mathcal{O}.\textsc{Recover}()$ before invoking any other operation on object $\mathcal{O}$ (if π crashes while executing $\mathcal{O}.\textsc{Recover}()$, it must execute $\mathcal{O}.\textsc{Recover}()$ again after restarting before invoking any other operation on $\mathcal{O}$); when π completes $\mathcal{O}.\textsc{Recover}()$, we deem $\mathcal{O}.op$ completed. If the usage pattern is observed, MRL guarantees that the crashed operation $\mathcal{O}.op$ either never takes effect or takes effect at some point between $\mathcal{O}.op$'s invocation and completion. Notice that the usage pattern allows π to perform any number of other operations on objects other than $\mathcal{O}$ upon restart and even allows for π to never perform any subsequent operation on $\mathcal{O}$; the only requirement is that π calls $\mathcal{O}.\textsc{Recover}()$ before calling any other method on $\mathcal{O}$.

MRL's relationship to Recoverable Linearizability (RL) and Nested-safe Recoverable Linearizability (NRL) can be understood as follows. Just like MRL, RL requires that a crashed operation by process π on object $\mathcal{O}$ either does not take effect at all or takes effect before π's next invocation of an operation on $\mathcal{O}$. However, RL does not specify a mechanism by which π's operation can complete after its crash and before its subsequent operation starts. MRL is similar, but uses the mechanism of the recover method to complete (or ensure non-completion of) crashed operations. The NRL paper uses recover methods. However, in the NRL model each method has its own recover method, recover methods take arguments, and it is assumed that upon a crash, a recover method is called with the same arguments as the corresponding method that failed and the recover method has access to a process specific persistent register which stores the program counter value right before the crash. In contrast to NRL, MRL has only a single recover method, and most importantly, makes no assumptions about method arguments or program counter values being supplied to restarted processes; it simply guarantees that objects remain consistent if the usage pattern is followed (however an implementation may follow it).

Friedman et al. [19] first made the observation that in addition to proper recovery, it is necessary for a process to be able to *detect* whether its crashed operation took effect, and if so, what its return value was. We ensure detectability of our operations in a method-based manner, by requiring that an object $\mathcal{O}$ provide an additional DETECT() method, which returns this information. We note that some operations, such as read or a failed CAS, can safely be repeated, regardless of whether they took effect [5, 7], whereas, a write or a successful CAS that changed the value of the object cannot be repeated safely. Our implementations provide a DETECT() method which guarantees that all unsafe-to-repeat operations are detected along with their responses, and that any operation that is not detected is safe to repeat. In particular, DETECT() returns a pair that satisfies the following property:

Method-based Detectability: If a process calls DETECT() twice—just before executing an operation and just after completing that (possibly crashed[3]) operation—and these successive calls to DETECT() return (d_1, r_1) and (d_2, r_2) respectively, then the following two conditions are satisfied:

1. If $d_2 > d_1$, then the operation took effect and its response is r_2.
2. Otherwise, $d_1 = d_2$ and the operation is safe to repeat.

A *durable* object is one that satisfies method-based recoverable linearizability and method-based detectability.

4 Handles for dynamic joining and space adaptivity

When a process calls a method to execute an operation *op*, the call is of the form $op(p, args)$, where *args* is a list of *op*'s arguments and p identifies the calling process. The methods use p to facilitate helping between processes. In many algorithms, the processes are given *pid*s from 1 to N, and p is the *pid* of the caller [5, 7]. In particular, p is used to index a pre-allocated helping array—in Ben-David et al.'s algorithm this helping array is of length N, one location per process being helped; in Attiya et al.'s algorithm this helping array is of length N^2, one location per helper-helpee pair. Helping plays a central role in detection, thus each process needs to have some area in memory where it can be helped; in fact, using the bit-complexity model, Ben-Baruch et al. proved that the space needed to support a detectable CAS object monotonically increases in the number of processes that access the object [6]. One of our goals in this paper however, is to design objects that can be accessed by a dynamically increasing set of processes, which precludes the use of pre-allocated fixed-size arrays that are indexed by process IDs.

To eliminate the use of arrays for helping, we introduce pointer based structures called *handles*. We use handles to enable dynamic joining and achieve space adaptivity. A handle is a pointer to a constant sized record. The implementation provides a CREATEHANDLE() method, which allocates memory for a new record and returns a pointer h to it called a

[3] Recall that a crashed operation by π *completes* when π finishes RECOVER() following the crash.

handle. This allocation can be via a persistent memory allocator [13] or any other means that ensures that handles are created in constant time and remembered across crashes without any memory leaks. When a process first wishes to access any of the implemented objects of a given type, it creates for itself a new handle by calling CreateHandle(). From that point on, whenever the process calls any method on any of the implemented objects of that type, it passes its handle h instead of its *pid*, and other processes help it via the handle. This mechanism of handles helps us realize dynamic joining because any number of new processes can join at any time by creating handles for themselves; since the memory per handle is constant, and only the subset of processes that wish to access the implementation need to create handles, the mechanism facilitates space adaptivity.

At first glance, replacing *pid*s with handles to achieve dynamic joining may seem like a simple level of indirection; however, this step actually poses a significant algorithmic challenge. As will become more clear in the next section, the challenge arises from the fact that known algorithms for durable primitives (including ours) must, in principle, store three pieces of information consistently using hardware that only supports double-width CAS. The three pieces to store are the implemented object's value, its sequence number, and the *pid*/handle of the process that last changed the object. (Intuitively, the sequence number helps styme ABA problems, while the handle facilitates helping.) Previous durable CAS algorithms either require the strong assumption that processes never attempt to CAS in the same value twice to avoid sequence numbers [5]; or assume that a *pid* and a sequence number can be packed into a single pointer-sized word [7]. Since handles are pointers, and the object's value can also be pointer-sized, we cannot pack all three pieces of information into a single double-width word. Our DurEC algorithm overcomes this challenge by storing the sequence number with the handle in one variable, and the same sequence number with the value in another variable, and cleverly coordinating between these two variables to create the illusion that all three pieces of information are stored and manipulated atomically together.

5 The DurEC Building Block

In this section, we implement the DurEC algorithm for durable external context non-writable LLSC using hardware CAS. This building block will be central to all of the writable implementations in the remainder of the paper.

Intuitive description of Algorithm DurEC: Each DurEC handle h is a reference to a record of two fields, *Val* and *DetVal*, and each DurEC object $\mathcal{O}$ is implemented from two hardware atomic CAS objects X and Y, where X is a pair consisting of a handle and a sequence number, and Y is a pair consisting of a sequence number and a value. The algorithm maintains the DurEC object $\mathcal{O}$'s state in Y, i.e., $\mathcal{O}.seq = Y.seq$ and $\mathcal{O}.val = Y.val$ at all times. This representation makes the implementation of ECLL and ECVL operations obvious: ECLL(h) simply returns Y and ECVL(h, s) returns whether $Y.seq = s$ (although ECLL and ECVL do not use the handle parameter h, for uniformity we let the handle be the first parameter of all object operations). The complexity lies in the ECSC(h, s, v) operation, which is implemented by the following sequence of steps:

1. If $Y.seq \neq s$, it means $\mathcal{O}.seq \neq s$, so the ECSC operation simply returns *false*. Otherwise, it embarks on the following steps, in an attempt to switch $\mathcal{O}.val$ to v and $\mathcal{O}.seq$ to a greater number.
2. Make v available for all by writing it in the *Val* field of the ECSC operation's handle h.
3. Pick a number $\hat{s}$ that is bigger than both $X.seq$ and $h.DetVal$. (The latter facilitates detection.)

4. Publish the operation's handle along with a greater sequence number by installing $(h, \hat{s})$ in X. If several ECSC operations attempt to install concurrently, only one will succeed. The successful one is the *installer* and the others are *hitchhikers*.
5. The installer and the hitchhikers work together to accomplish two missions, the first of which is to increase the installer's $DetVal$ field to the number in $X.seq$. This increase in the $DetVal$ field of its handle enables the installer to detect that it installed, even if the installer happens to crash immediately after installing.
6. The second mission is to forward the installer's operation to Y. Since Y is where the DurEC object's state is held, the installer's operation takes effect only when it is reflected in Y's state. Towards this end, everyone reads the installer's value v, made available in the Val field of the installer's handle back at Step (2), and attempts to switch $Y.val$ to v, simultaneously increasing $Y.seq$ so that it catches up with $X.seq$. Since all operations attempt this update of Y, someone (not necessarily the installer) will succeed. At this point, $X.seq = Y.seq$ and $Y.val = v$, which means that the installer's value v has made its way to $\mathcal{O}.val$. So, the point where Y is updated becomes the linearization point for the installer's successful ECSC operation. The hitchhikers are linearized immediately after the installer, which causes their ECSC operations to "fail"—return *false*, without changing $\mathcal{O}$'s state—thereby eliminating the burden of detecting these operations.
7. If the installer crashes after installing, upon restart, in the Recover method, it does the forwarding so that the two missions explained above are fulfilled.
8. With the above scheme, all ECSC, ECLL, and ECVL operations, except those ECSC operations that install, are safe to repeat and hence, don't need detection. Furthermore, for each installing ECSC operation, the above scheme ensures that the $DetVal$ field of the installer's handle is increased, thereby making the operation detectable.

The formal algorithm is presented in Figure 1. The correspondence between the lines of the algorithm and the steps above is as follows. Lines 6 and 7 implement Steps 1 and 2, respectively. Steps 3 and 4, where the operation attempts to become the installer, are implemented by Lines 8 to 10. The operation becomes the installer if and only if the CAS at Line 10 succeeds, which is reflected in the boolean return value r. The Forward method is called at Line 11 to accomplish the two missions described above. The first three lines of Forward (Lines 13 to 15) implement the first mission of increasing the $DetVal$ field of the installer's handle to $X.seq$ (Step 5). Line 13, together with Lines 16 to 19, implement the second mission of forwarding the operation to Y (Step 6). The if-condition and the CAS' arguments at Line 18 ensure that Y is changed only if $Y.seq$ lags behind $X.seq$ and, if it lags behind, it catches up and $Y.val$ takes on the installer's value. The Recover method simply forwards at Line 20, as explained in Step 7. The detect method returns at Line 22 the value in the handle's $DetVal$ field, as explained in Step 8, along with *true* (since only successful ECSC operations are detected).

The theorem below summarizes the result:

▶ **Theorem 1.** *Algorithm DurEC satisfies the following properties:*
1. *The objects implemented by the algorithm are durable, i.e., they satisfy method-based recoverable linearizability and method-based detectability (with respect to the sequential specification of External Context LLSC).*
2. *All object operations including Recover are wait-free and run in constant time.*
3. *The algorithm supports dynamic joining: a new process can join in at any point in a run (by calling CreateHandle) and start creating DurEC objects or accessing existing DurEC objects.*

4. *The space requirement is $O(m+n)$, where m is the actual number of* DurEC *objects created in the run, and n is the actual number of processes that have joined in in a run.*

▶ Remark. It is interesting to note that while DurEC is *recoverably linearizable*, it is not *strictly lineariable*, i.e., operations may linearize after a process crashes (but before its next successful completion of Recover()). In particular, if a process crashes after successfully CASing into X and before changing Y, then this operation has not taken effect before the crash, but is guaranteed to take effect in the future (i.e., after the crash).

Algorithm 1 : The DurEC class for Durable, External Context nW-LLSC objects.

```
class DurEC:
    instance variable  (handle*, int)  X      ▷ $X$ is a pair $(X.hndl, X.seq)$ stored in NVM
    instance variable  (int, int)  Y          ▷ $Y$ is a pair $(Y.seq, Y.val)$ stored in NVM

    struct handle {
        int DetVal
        int Val
    }

    static procedure CreateHandle()
1:      return new $handle\{DetVal = 0\}$      ▷ DetVal and Val in NVM; Val arbitrarily initialized

    constructor DurEC(int initval)
2:      $X \leftarrow (null, 0)$
3:      $Y \leftarrow (0, initval)$

    procedure ECLL(handle* h)
4:      return $Y$

    procedure ECVL(handle* h, int s)
5:      return $Y.seq = s$

    procedure ECSC(handle* h, int s, int v)
6:      if $Y.seq \neq s$ then return false
7:      $h.Val \leftarrow v$
8:      $\hat{h} \leftarrow X.hndl$
9:      $\hat{s} \leftarrow \max(\hat{h}.DetVal, s) + 1$
10:     $r \leftarrow$ CAS$(X, (\hat{h}, s), (h, \hat{s}))$
11:     forward(h)
12:     return $r$

    procedure forward(handle* h)
13:     $x \leftarrow X$
14:     $\hat{s} \leftarrow x.hndl.DetVal$
15:     if $\hat{s} < x.seq$ then CAS$(x.hndl.DetVal, \hat{s}, x.seq)$
16:     $\hat{v} \leftarrow x.hndl.Val$
17:     $y \leftarrow Y$
18:     if $y.seq < x.seq$ then CAS$(Y, y, (x.seq, \hat{v}))$
19:     return

    procedure Recover(handle* h)
20:     forward(h)
21:     return

    static procedure Detect(handle* h)
22:     return $(h.DetVal, true)$
```

6 DurECW and DuraLL: durable Writable LLSC implementations

Using DurEC, we design the *writable* external context LLSC implementation DurECW in this section. With DurECW in hand, we obtain our standard durable writable-LLSC implementation DuraLL easily, by simply rolling the context into the object.

6.1 Intuitive description of Algorithm DurECW

A DurECW object $\mathcal{O}$ supports the write operation, besides ECSC, for changing the object's state. Unlike a ECSC(h, s, v) operation, which returns without changing $\mathcal{O}$'s state when $\mathcal{O}.context \neq s$, a WRITE$(h, v)$ must get v into $\mathcal{O}.val$ unconditionally. In the DurECW algorithm, ECSC() operations help WRITE() operations and prevent writes from being blocked by a continuous stream of successful ECSC() operations.

Each DurECW object $\mathcal{O}$ is implemented from two DurEC objects, $\mathcal{W}$ and $\mathcal{Z}$, each of which holds a pair, where the first component is a sequence number *seq*, and the second component is a pair consisting of a value *val* and a bit *bit*. Thus, $\mathcal{W} = (\mathcal{W}.seq, (\mathcal{W}.val, \mathcal{W}.bit))$ and $\mathcal{Z} = (\mathcal{Z}.seq, (\mathcal{Z}.val, \mathcal{Z}.bit))$.

The DurECW handle h consists of two DurEC handles, $h.Critical$ and $h.Casual$. The use of two DURTEC handles allows us to implement detectability. In particular, if DETECT(h) is called on a DURECW object, only the detect value (*DetVal*) of $h.Critical$ is returned. So intuitively, when a DurECW operation α calls methods on $\mathcal{W}$ or $\mathcal{Z}$, it uses $h.Critical$ only if a successful call will make its own ECSC() or WRITE() operation visible. In all other cases α uses $h.Casual$.

The algorithm maintains the DurECW object $\mathcal{O}$'s state in $\mathcal{Z}$, i.e., $\mathcal{O}.seq = \mathcal{Z}.seq$ and $\mathcal{O}.val = \mathcal{Z}.val$ at all times. This representation makes the implementation of $\mathcal{O}$.ECLL() and $\mathcal{O}$.ECVL() operations obvious: $\mathcal{O}$.ECLL(h) simply returns $(\mathcal{Z}.seq, \mathcal{Z}.val)$ and ECVL(h, s) returns whether $\mathcal{Z}.seq = s$. The complexity lies in the implementation of $\mathcal{O}$.WRITE(h, v) and $\mathcal{O}$.ECSC(h, s, v) operations, which coordinate their actions using $\mathcal{W}.bit$ and $\mathcal{Z}.bit$. A write operation flips the $\mathcal{W}.bit$ to announce to the ECSC operations that their help is needed to push the write into $\mathcal{Z}$; once the write is helped, the $\mathcal{Z}.bit$ is flipped to announce that help is no longer needed. We maintain the invariant that $\mathcal{W}.bit \neq \mathcal{Z}.bit$ if and only if a write needs help.

A WRITE(h, v) operation α consists of the following steps.

(W1) The operation α reads $\mathcal{W}$ and $\mathcal{Z}$ to determine if some write operation is already waiting for help. If not, then α installs its write into $\mathcal{W}$ by setting $\mathcal{W}.val$ to v and flipping $\mathcal{W}.bit$. If several write operations attempt to install concurrently, only one will succeed. The successful one is the *installer* and the others are *hitchhikers*.

(W2) Once a write operation is installed, all processes—installer, hitchhiker, and the ECSC operations—work in concert to forward the installer's operation to $\mathcal{Z}$. Since $\mathcal{Z}$ is where the DurECW object's state is held, the installer's operation takes effect only when it is reflected in $\mathcal{Z}$'s state. Towards this end, everyone attempts to transfer the installer's value from $\mathcal{W}$ to $\mathcal{Z}$. However, a stale ECSC operation, which was poised to execute its ECSC operation on $\mathcal{Z}$, might update $\mathcal{Z}$, causing the transfer to fail in moving the installer's value from $\mathcal{W}$ to $\mathcal{Z}$. So, a transfer is attempted the second time. The earlier success by the poised ECSC operation causes any future attempts by similarly poised operations to fail. Consequently, the installer's write value gets moved to $\mathcal{Z}$ by the time the second transfer attempt completes. The point where the move to $\mathcal{Z}$ occurs becomes the linearization point for the installer's write operation. We linearize the writes by the hitchhikers immediately before the installer, which makes their write operations to be overwritten immediately by the installer's write, without anyone ever witnessing their writes. Hence, there is no need to detect these writes: if a hitchhiker crashes during its write, the operation can be safely repeated.

(W3) If the installer crashes after installing, upon restart, in the Recover method, it does the forwarding so that its install moves to $\mathcal{Z}$ and its write operation gets linearized.

Algorithm 2 The DurECW class for Durable External Context W-LLSC objects.

```
class DurECW:
    instance variable DurEC $\mathcal{W}$        ▷ $\mathcal{W}$ holds a pair $(\mathcal{W}.seq, (\mathcal{W}.val, \mathcal{W}.bit))$
    instance variable DurEC $\mathcal{Z}$        ▷ $\mathcal{Z}$ holds a pair $(\mathcal{Z}.seq, (\mathcal{Z}.val, \mathcal{Z}.bit))$

    struct handle {
        DurEC.handle Critical
        DurEC.handle Casual
    }

    static procedure CreateHandle()
1:      return new $handle\{Critical \leftarrow$ DurEC.CreateHandle(), $Casual \leftarrow$ DurEC.CreateHandle()$\}$

    procedure DurECW($initval$)
2:      $\mathcal{W} \leftarrow$ DurEC($(0, 0)$)
3:      $\mathcal{Z} \leftarrow$ DurEC($(initval, 0)$)

    procedure ECLL(handle* $h$)
4:      $z \leftarrow \mathcal{Z}$.ECLL($h.Casual$)
5:      return $(z.seq, z.val)$

    procedure ECVL(handle* $h$, int $s$)
6:      return $\mathcal{Z}$.ECVL($h.Casual, s$)

    procedure ECSC(handle* $h$, int $s$, int $v$)
7:      $z \leftarrow \mathcal{Z}$.ECLL($h.Casual$)
8:      if $s \neq z.seq$ return $false$
9:      transfer-write($h$)
10:     $r \leftarrow \mathcal{Z}$.ECSC($h.Critical, s, (v, z.bit)$)
11:     return $r$

    procedure Write(handle* $h$, int $v$)
12:     $w \leftarrow \mathcal{W}$.ECLL($h.Casual$)
13:     $z \leftarrow \mathcal{Z}$.ECLL($h.Casual$)
14:     if $z.bit = w.bit$ then $\mathcal{W}$.ECSC($h.Critical, w.seq, (v, 1 - w.bit)$)
15:     transfer-write($h$)
16:     transfer-write($h$)
17:     return $true$

    procedure transfer-write(handle* $h$)
18:     $\hat{z} \leftarrow \mathcal{Z}$.ECLL($h.Casual$)
19:     $\hat{w} \leftarrow \mathcal{W}$.ECLL($h.Casual$)
20:     if $\hat{z}.bit \neq \hat{w}.bit$ then $\mathcal{Z}$.ECSC($h.Casual, \hat{z}.seq, (\hat{w}.val, \hat{w}.bit)$)

    procedure Recover(handle* $h$)
21:     $\mathcal{W}$.Recover($h.Critical$)
22:     $\mathcal{Z}$.Recover($h.Critical$)
23:     $\mathcal{W}$.Recover($h.Casual$)
24:     $\mathcal{Z}$.Recover($h.Casual$)
25:     transfer-write($h$)
26:     transfer-write($h$)

    static procedure Detect(handle* $h$)
27:     return DurEC.Detect($h.Critical$)
```

An ECSC(h, s, v) operation α consists of the following steps.

(S1) α performs an ECLL() to determine whether the context in $\mathcal{O}$ matches s. If not, it can fail early and return *false*.

(S2) If a Write() is already in $\mathcal{W}$ and waiting for help to be transferred to $\mathcal{Z}$, α is obligated to help that write before attempting its SC (to prevent the write from being blocked by a chain of successful ECSC() operations). So it attempts a transfer from $\mathcal{W}$ to $\mathcal{Z}$.

(S3) Finally α executes an ECSC() on $\mathcal{Z}$ in an attempt to make its own operation $\mathcal{O}$ take effect.

The algorithm is formally presented in Algorithm 2. In the algorithm, Lines 12-14 implement step W1 and Lines 15, 16 implement step W2. Step S1 is implemented by Lines 7, 8, step S2 by 9 and S3 by 10 and 11. Note that the ECSC() on line 10 takes care to not change $\mathcal{Z}.bit$. This ensures that the helping mechanism for writes implemented via $\mathcal{W}.bit$ and $\mathcal{Z}.bit$ is not disturbed. The ECSC() operation at Line 14 uses the handle $h.Critical$ because its success implies that the operation is an installer and hence will be a visible write when it linearizes. Similarly the ECSC() on $\mathcal{Z}$ at Line 10 uses $h.Critical$ because its success makes the ECSC() on $\mathcal{O}$ visible.

If a WRITE() or a ECSC() method crashes while executing an operation on $\mathcal{W}$ or $\mathcal{Z}$, upon restart, Lines 21 to 24 of RECOVER() ensure that $\mathcal{W}$.RECOVER() or $\mathcal{Z}$.RECOVER() is executed before any other operation is executed on $\mathcal{W}$ or $\mathcal{Z}$ (the relative order of lines 21-24 is unimportant). Consequently, the durable objects $\mathcal{W}$ and $\mathcal{Z}$ behave like atomic EC objects.

The theorem below summarizes the result:

▶ **Theorem 2.** *Algorithm DURECW satisfies the following properties:*
1. *The objects implemented by the algorithm are durable, i.e., they satisfy method-based recoverable linearizability and method-based detectability (with respect to the sequential specification of External Context Writable-LLSC).*
2. *All object operations including Recover are wait-free and run in constant time.*
3. *The algorithm supports dynamic joining: a new process can join in at any point in a run (by calling CreateHandle) and start creating DURECW objects or accessing existing DURECW objects.*
4. *The space requirement is $O(m + n)$, where m is the actual number of DURECW objects created in the run, and n is the actual number of processes that have joined in in a run.*

6.2 The DuraLL Algorithm

Given the durable EC W-LLSC object DURECW, rolling the context into the implementation to produce a durable standard W-LLSC object is simple. Each of our implemented DURALL objects simply maintains a single DURECW object X. The handle of the DURALL object simply maintains a single DURECW handle to operate on X, and a hashmap, *cntxts*, that maps objects to contexts.

We present the DURALL code as Algorithm 4 in the Appendix A. The LL() operation on a DURALL object by handle h simply performs an ECLL() on X and stores the returned context in $h.cntxts$ under the key *self* (which is the reference of the current object). Correspondingly, VL() retrieves the context from $h.cntxts$, and uses it to perform an ECVL() on X. The SC() operation also retrieves the context and performs an ECSC() on the internal object, but then cleverly removes the key corresponding to the current object from $h.cntxts$, since, regardless of whether the SC() succeeds, the stored context is bound to be out-of-date. The WRITE() operation does not need a context, so it simply writes to X, but also cleverly removes the current object's key from $h.cntxts$ to save some space. In order to be space-efficient, RECOVER() also removes the current object from $h.cntxts$ if the context stored for the object is out-of-date. Since DURALL is just a wrapper around DURECW, its DETECT() operation simply returns the result of detecting DURECW.

▶ **Theorem 3.** *Algorithm DURALL satisfies the following properties:*
1. *The objects implemented by the algorithm are durable, i.e., they satisfy method-based recoverable linearizability and method-based detectability (with respect to the sequential specification of Writable LLSC).*
2. *All object operations including Recover are wait-free and run in constant time.*

3. *The algorithm supports dynamic joining: a new process can join in at any point in a run (by calling CreateHandle) and start creating* DuraLL *objects or accessing existing* DuraLL *objects.*
4. *The space requirement is $O(m+n+C)$, where m is the actual number of* DuraLL *objects created in the run, n is the actual number of processes that have joined in in a run, and C is the number of "contexts" stored across all objects.*

7 DuraCAS: a durable implementation of Writable CAS

We present in Figure 3 Algorithm DuraCAS, which implements a durable writable CAS object $\mathcal{O}$ from two DurEC objects, $\mathcal{W}$ and $\mathcal{Z}$. The algorithm bears a lot of similarity to Algorithm DurECW of the previous section. In fact, DuraCAS has only three extra lines. For readability, we starred their line numbers (Lines **6***, **10***, and **13***) and kept the line numbers the same for the common lines.

The ideas underlying this algorithm are similar to DurECW, so we explain here only the three differences: (1) Lines 7 to 10 are executed only once in Algorithm DurECW, but are repeated twice in the current algorithm; (2) Line 8 differs in the two algorithms; and (3) Line 13* is introduced in the current algorithm.

The change in Line 8 accounts for the fact that the success of a Cas() operation depends on the value in $\mathcal{O}$ rather than the context. If the value in $\mathcal{O}$ (and therefore $\mathcal{Z}$) is different from *old* at Line 7, the CAS returns *false* (and linearizes at Line 7). If $\mathcal{O}.val = old$ and the CAS does not plan to change the value (i.e., $old = new$) it returns *true* without changing $\mathcal{Z}$.

To understand why Lines 7 to 10 are repeated in the current algorithm, consider the following scenario. A handle h executes $\mathcal{O}.CAS(h, old, new)$, where $old \neq new$. When h executes Line 7, $\mathcal{Z}$'s value is *old*, so $z.val$ gets set to *old* at Line 7. Handle h progresses to Line 10, but before it executes Line 10, some handle h' invokes $\mathcal{O}.\text{Write}(h', old)$ and executes it to completion, causing $\mathcal{Z}.seq$ to take on a value greater than $z.seq$. Handle h now executes the ECSC at Line 10 and fails since $\mathcal{Z}.seq \neq z.seq$. If h acts as it did in Algorithm DurECW, h would complete its $\mathcal{O}.CAS(h, old, new)$ operation, returning *false*. However, *false* is an incorrect response by the specification of CAS because $\mathcal{O}.val = old$ for the full duration of the operation $\mathcal{O}.CAS(h, old, new)$. To overcome this race condition, h repeats Lines 7 to 10.

If the same race condition repeats each time h repeats Lines 7 to 10, the method $\mathcal{O}.CAS$ would not be wait-free. Line 13* is introduced precisely to prevent this adverse possibility. When a handle h' executes Lines 12 to 14 of $\mathcal{O}.\text{Write}(h', v)$ in the previous DurECW algorithm, h' would always try to install its value v in $\mathcal{W}$ (at Line 14) and later move it to $\mathcal{Z}$, thereby increasing $\mathcal{Z}.seq$ and causing concurrent $\mathcal{O}.\text{ECSC}()$ operations to fail. This was precisely what we wanted because the specification of an SC operation requires that if *any* $\mathcal{O}.\text{Write}()$ takes effect, regardless of what value it writes in $\mathcal{O}$, it must change $\mathcal{O}.context$ and thus cause concurrent $\mathcal{O}.\text{ECSC}()$ operations to fail. The situation however, is different when implementing $\mathcal{O}.CAS$, where a $\mathcal{O}.\text{Write}()$ that does not change the value in $\mathcal{O}$ should not cause a concurrent $\mathcal{O}.CAS$ to fail. Hence, if a $\mathcal{O}.\text{Write}(h', v)$ operation is writing the same value as $\mathcal{O}$'s current value, then it should simply return (since $\mathcal{O}.val$ already has v) and, importantly, not change $\mathcal{Z}.seq$ (because changing $\mathcal{Z}.seq$ would cause any concurrent *CAS* operation to fail). Line 13* implements precisely this insight by ensuring that two $\text{Write}(-, v)$ operations both change $\mathcal{Z}$ only if there is some $\text{Cas}(-, v, v')$ or $\text{Write}(-, v')$ operation that changes $\mathcal{Z}$ in between (for some $v' \neq v$).

The theorem below summarizes the result:

Algorithm 3 The DuraCAS class for Durable, Writable-CAS objects.

```
class DuraCAS:
    instance variable DurEC $\mathcal{W}$        ▷ $\mathcal{W}$ holds a pair $(\mathcal{W}.seq, (\mathcal{W}.val, \mathcal{W}.bit))$
    instance variable DurEC $\mathcal{Z}$        ▷ $\mathcal{Z}$ holds a pair $(\mathcal{Z}.seq, (\mathcal{Z}.val, \mathcal{Z}.bit))$

    struct handle {
        DurEC.handle* Critical
        DurEC.handle* Casual
    }

    static procedure CreateHandle()
1:      return new handle{Critical ← DurEC.CreateHandle(), Casual ← DurEC.CreateHandle()}

    procedure DuraCAS(int initval)
2:      $\mathcal{W} \leftarrow$ DurEC((0, 0))
3:      $\mathcal{Z} \leftarrow$ DurEC((initval, 0))

    procedure Read(handle* h)
4:      $z \leftarrow \mathcal{Z}$.ECLL(h.Casual)
5:      return z.val

6:

    procedure CAS(handle* h, int old, int new)
6*:      for $i \leftarrow 1$ to 2
7:          $z \leftarrow \mathcal{Z}$.ECLL(h.Casual)
8:          if $z.val \neq old$ then return false else if $old = new$ then return true
9:          transfer-write(h)
10:         if $\mathcal{Z}$.ECSC(h.Critical, z.seq, (new, z.bit)) then
10*:            return true
11:     return false

    procedure Write(handle* h, int v)
12:     $w \leftarrow \mathcal{W}$.ECLL(h.Casual)
13:     $z \leftarrow \mathcal{Z}$.ECLL(h.Casual)
13*:    if $z.val = v$ then return ack
14:     if $z.bit = w.bit$ then $\mathcal{W}$.ECSC(h.Critical, w.seq, (v, 1 − w.bit))
15:     transfer-write(h)
16:     transfer-write(h)
17:     return ack

    procedure transfer-write(handle* h)
18:     $\hat{z} \leftarrow \mathcal{Z}$.ECLL(h.Casual)
19:     $\hat{w} \leftarrow \mathcal{W}$.ECLL(h.Casual)
20:     if $\hat{z}.bit \neq \hat{w}.bit$ then $\mathcal{Z}$.ECSC(h.Casual, $\hat{z}$.seq, ($\hat{w}$.val, $\hat{w}$.bit))

    procedure Recover(handle* h)
21:     $\mathcal{W}$.Recover(h.Critical)
22:     $\mathcal{Z}$.Recover(h.Critical)
23:     $\mathcal{W}$.Recover(h.Casual)
24:     $\mathcal{Z}$.Recover(h.Casual)
25:     transfer-write(h)
26:     transfer-write(h)

    static procedure Detect(handle* h)
27:     return DurEC.Detect(h.Critical)
```

▶ **Theorem 4.** *Algorithm DuraCAS satisfies the following properties:*

1. *The objects implemented by the algorithm are durable, i.e., they satisfy method-based recoverable linearizability and method-based detectability (with respect to the sequential specification of Writable CAS).*
2. *All object operations including Recover are wait-free and run in constant time.*
3. *The algorithm supports dynamic joining: a new process can join in at any point in a run (by calling CreateHandle) and start creating DuraCAS objects or accessing existing*

DuraCAS objects.

4. *The space requirement is $O(m + n)$, where m is the actual number of DuraCAS objects created in the run, and n is the actual number of processes that have joined in in a run.*

8 Discussion and Remarks

In this paper, we have designed constant time implementations for durable CAS and LLSC objects. To our knowledge, DuraCAS is the first CAS implementation to allow for dynamic joining. DuraCAS also has state-of-the-art space complexity—allowing adaptivity and requiring only constant space per object and per process that actually accesses the protocol—and is writable. To our knowledge, ours are the first implementations of durable LLSC objects. LLSC objects are universal and ABA-free, thus we believe that the dynamically joinable LLSC implementations in this paper will be useful in the construction of several more complex durable objects. The external context variant of LLSC is particularly space efficient, making it a powerful building block for concurrent algorithms; we witnessed this property even in the constructions of this paper, where the EC nW-LLSC object DurEC served as the primary building block for all our other implementations, including our EC W-LLSC implementation DurECW and its direct descendent DuraLL (for W-LLSC). All the implementations in this paper were enabled by handles—a pointer-based mechanism we introduced to enable threads created on-the-fly to access our implementations. We believe that along with the specific implementations of this paper, the use of handles as an algorithmic tool can play an important role in the design of future durable algorithms.

We end with two open problems. Handles enable dynamic joining, but once a handle h is used, any other process can have a stale pointer to h that may be dereferenced at any point in the future. A mechanism for enabling space adaptivity for both dynamic joining and *dynamic leaving*, which would enable a process to reclaim its entire memory footprint once it is done using a durable implementation is our first open problem. Our second open problem is to prove (or disprove) an $\Omega(m + n)$ space lower bound for supporting m objects for n processes for any durable CAS or durable LLSC type.

References

1 Zahra Aghazadeh, Wojciech Golab, and Philipp Woelfel. Making objects writable. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, page 385–395, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2611462.2611483.

2 Marcos K. Aguilera and Svend Frølund. Strict linearizability and the power of aborting. In *techreport*, 2003.

3 James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, page 184–193, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/224964.224985.

4 Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios 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 262–277, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503221.3508444.

5 Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, page 7–16, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3212734.3212753.

6 Ohad Ben-Baruch, Danny Hendler, and Matan 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, page 11–20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3382734.3405725.

7 Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, page 253–264, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3323165.3323187.

8 Naama Ben-David, Michal Friedman, and Yuanhao Wei. Brief announcement: Survey of persistent memory correctness conditions. In Christian Scheideler, editor, *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA*, volume 246 of *LIPIcs*, pages 41:1–41:4. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.DISC.2022.41.

9 Ryan Berryhill, Wojciech M. Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In Emmanuelle Anceaume, Christian Cachin, and Maria Gradinariu Potop-Butucaru, editors, *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France*, volume 46 of *LIPIcs*, pages 20:1–20:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.OPODIS.2015.20.

10 Guy E. Blelloch, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. The parallel persistent memory model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, page 247–258, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3210377.3210381.

11 Guy E. Blelloch and Yuanhao Wei. LL/SC and atomic copy: Constant time, space efficient implementations using only pointer-width CAS. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 5:1–5:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.5.

12 Philip Bohannon, Daniel Lieuwen, Avi Silberschatz, S. Sudarshan, and Jacques Gava. Recoverable User-Level Mutual Exclusion. In *In Proc. 7th IEEE Symposium on Parallel and Distributed Processing*, 1995.

13 Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. Understanding and optimizing persistent memory allocation. In *Proceedings*

of the 2020 ACM SIGPLAN International Symposium on Memory Management, ISMM 2020, page 60–73, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3381898.3397212.

14 David Yu Cheng Chan and Philipp Woelfel. Recoverable mutual exclusion with constant amortized RMR complexity from standard primitives. In *ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 2020. doi:10.1145/3382734.3405736.

15 David Yu Cheng Chan and Philipp Woelfel. Tight lower bound for the RMR complexity of recoverable mutual exclusion. In *ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 2021. doi:10.1145/3465084.3467938.

16 D. Dechev, P. Pirkelbauer, and B. Stroustrup. Understanding and effectively preventing the aba problem in descriptor-based lock-free designs. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 185–192, 2010. doi:10.1109/ISORC.2010.10.

17 Sahil Dhoked and Neeraj Mittal. An adaptive approach to recoverable mutual exclusion. In *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*. ACM, 2020. doi:10.1145/3382734.3405739.

18 Simon Doherty, Maurice Herlihy, Victor Luchangco, and Mark Moir. Bringing practical lock-free synchronization to 64-bit applications. In Soma Chaudhuri and Shay Kutten, editors, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, pages 31–39. ACM, 2004. doi:10.1145/1011767.1011773.

19 Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In Andreas Krall and Thomas R. Gross, editors, *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, pages 28–40. ACM, 2018. doi:10.1145/3178487.3178490.

20 Wojciech M. Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 211–220. ACM, 2017. doi:10.1145/3087801.3087819.

21 Wojciech M. Golab and Aditya Ramaraju. Recoverable mutual exclusion: [extended abstract]. In George Giakkoupis, editor, *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 65–74. ACM, 2016. doi:10.1145/2933057.2933087.

22 Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. Efficient multi-word compare and swap. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 4:1–4:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.4.

23 Rachid Guerraoui and Ron R. Levy. Robust emulations of shared memory in a crash-recovery model. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, ICDCS '04, page 400–407, USA, 2004. IEEE Computer Society.

24 happyware.com. Supermicro 8-socket intel xeon 7u rack server. https://happyware.com/uk-en/supermicro/sys-7089p-tr4t. Accessed: August 1, 2022.

25 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

26 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

27 Intel. Intel optane technology, 2020. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

28 Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, volume 9888 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2016. doi:10.1007/978-3-662-53426-7_23.

29 Prasad Jayanti. A complete and constant time wait-free implementation of cas from ll/sc and vice versa. In *Proceedings of the 12th International Symposium on Distributed Computing*, DISC '98, page 216–230, Berlin, Heidelberg, 1998. Springer-Verlag.

30 Prasad Jayanti, Siddhartha Jayanti, and Sucharita Jayanti. Durable algorithms for writable ll/sc and cas with dynamic joining, 2023. arXiv:2302.00135.

31 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. A recoverable mutex algorithm with sub-logarithmic rmr on both cc and dsm. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 177–186, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331634.

32 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. Constant rmr system-wide failure resilient durable locks with dynamic joining. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '23, page 227–237, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3558481.3591100.

33 Prasad Jayanti, Siddhartha V. Jayanti, and Anup Joshi. Optimal recoverable mutual exclusion using only FASAS. In Andreas Podelski and François Taïani, editors, *Networked Systems - 6th International Conference, NETYS 2018, Essaouira, Morocco, May 9-11, 2018, Revised Selected Papers*, volume 11028 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 2018. doi:10.1007/978-3-030-05529-5_13.

34 Prasad Jayanti, Siddhartha Visveswara Jayanti, and Sucharita Jayanti. Brief announcement: Efficient recoverable writable-cas. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*, PODC '23, page 366–369, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3583668.3594592.

35 Prasad Jayanti and Anup Joshi. Recoverable FCFS mutual exclusion with wait-free recovery. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 30:1–30:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.DISC.2017.30.

36 Prasad Jayanti and Anup Joshi. Recoverable mutual exclusion with abortability. In Mohamed Faouzi Atig and Alexander A. Schwarzmann, editors, *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers*, volume 11704 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2019. doi:10.1007/978-3-030-31277-0_14.

37 Prasad Jayanti and Srdjan Petrovic. Efficient and practical constructions of ll/sc variables. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, PODC '03, page 285–294, New York, NY, USA, 2003. Association for Computing Machinery. doi:10.1145/872035.872078.

38 Prasad Jayanti and Srdjan Petrovic. Efficiently implementing LL/SC objects shared by an unknown number of processes. In Ajit Pal, Ajay D. Kshemkalyani, Rajeev Kumar, and Arobinda Gupta, editors, *Distributed Computing - IWDC 2005, 7th International Workshop, Kharagpur, India, December 27-30, 2005, Proceedings*, volume 3741 of *Lecture Notes in Computer Science*, pages 45–56. Springer, 2005. doi:10.1007/11603771_5.

39 Nan Li and Wojciech M. Golab. Detectable sequential specifications for recoverable shared objects. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, pages 29:1–29:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.DISC.2021.29.

40 Maged M. Michael. Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In Rachid Guerraoui, editor, *Distributed Computing, 18th International Con-*

ference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings, volume 3274 of *Lecture Notes in Computer Science*, pages 144–158. Springer, 2004. doi:10.1007/978-3-540-30186-8_11.

41 Aditya Ramaraju. RGLock: Recoverable mutual exclusion for non-volatile main memory systems. Master's thesis, University of Waterloo, 2015. URL: https://uwspace.uwaterloo.ca/handle/10012/9473.

42 Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. Easy lock-free indexing in non-volatile memory. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 461–472. IEEE Computer Society, 2018. doi:10.1109/ICDE.2018.00049.

A DuraLL Implementation

Algorithm 4 The DURALL class for Durable Writable-LLSC objects.

```
class DURALL:
    instance variable DurECW $X$                    ▷ $X$ holds the central EC W-LLSC object.

    struct $handle$ {
        DurECW.handle $ECWH$
        HashMap (DuraLL $\to$ int) $cntxts$
    }

    static procedure CREATEHANDLE()
        return new $handle\{ECWH \leftarrow$ DURECW.CREATEHANDLE()$, cntxts \leftarrow$ HashMap(DuraLL $\to$ int)$\}$

    procedure DURALL($initval$)
        $X \leftarrow$ DURECW($initval, 0$)

    procedure LL(handle* $h$)
        $x \leftarrow X$.ECLL($h.ECWH$)
        $h.cntxts(self) \leftarrow x.seq$
        return $x.val$

    procedure VL(handle* $h$)
        if $self \notin h.cntxts.keys$ then return $false$
        return $X$.ECVL($h.ECWH, h.cntxts(self)$)

    procedure SC(handle* $h$, int $val$)
        if $self \notin h.cntxts.keys$ then return $false$
        $r \leftarrow X$.ECSC($h.ECWH, h.cntxts(self), val$)
        $h.cntxts$.REMOVE($self$)
        return $r$

    procedure WRITE(handle* $h$, int $val$)
        $X$.WRITE($h.ECWH, val$)
        $h.cntxts$.REMOVE($self$)
        return $true$

    procedure RECOVER(handle* $h$)
        $X$.RECOVER($h.ECWH$)
        if $self \in h.cntxts.keys$ then
            if $\neg X$.ECVL($h.ECWH, h.cntxts(self)$) then $h.context$.REMOVE($self$)

    static procedure DETECT(handle* $h$)
        return DURECW.DETECT($h.ECWH$)
```