Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model

Joseph Izraelevitz(✉), Hammurabi Mendes, and Michael L. Scott

University of Rochester, Rochester, NY 14627-0226, USA
{jhi1,hmendes,scott}@cs.rochester.edu

Abstract. This paper provides a theoretical and practical framework for crash-resilient data structures on a machine with persistent (nonvolatile) memory but transient registers and cache. In contrast to certain prior work, but in keeping with "real world" systems, we assume a full-system failure model, in which all transient state (of all processes) is lost on a crash. We introduce the notion of *durable linearizability* to govern the safety of concurrent objects under this failure model and a corresponding relaxed, buffered variant which ensures that the persistent state in the event of a crash is consistent but not necessarily up to date.

At the implementation level, we present a new "memory persistency model," *explicit epoch persistency*, that builds upon and generalizes prior work. Our model captures both hardware buffering and fully relaxed consistency, and subsumes both existing and proposed instruction set architectures. Using the persistency model, we present an automated transform to convert any linearizable, nonblocking concurrent object into one that is also durably linearizable. We also present a design pattern, analogous to linearization points, for the construction of other, more optimized objects. Finally, we discuss generic optimizations that may improve performance while preserving both safety and liveness.

1 Introduction

Current industry projections indicate that nonvolatile, byte-addressable memory (NVM) will become commonplace over the next few years. While the availability of NVM suggests the possibility of keeping persistent data in main memory (not just in the file system), the fact that recent updates to registers and cache may be lost during a power failure means that the data in main memory, if not carefully managed, may not be consistent at recovery time.

Maintaining a consistent state in NVM requires special care to order main memory updates. Several groups have designed libraries to support such ordering using failure atomic updates, via either a transactional memory interface [6,7,20,26] or one inferred from mutex synchronization [5,17]. Others

This work was supported in part by NSF grants CCF-0963759, CCF-1116055, CNS-1319417, CCF-1422649, and CCF-1337224, and by support from the IBM Canada Centres for Advanced Study.

C. Gavoille and D. Ilcinkas (Eds.): DISC 2016, LNCS 9888, pp. 313–327, 2016.
DOI: 10.1007/978-3-662-53426-7_23

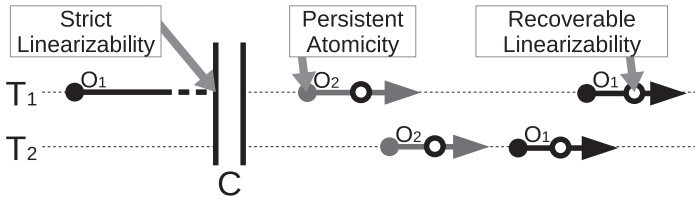

Fig. 1. Linearization bounds for interrupted operations under thread reuse failure model. Displayed is a concurrent abstract (operation-level) history of two threads (T_1 and T_2) on two objects (O_1 and O_2); linearization points are shown as circles. These correctness conditions differ in the deadline for linearization for a pending operation interrupted by a crash (T_1's first operation). *Strict linearizability* [1] requires that the pending operation linearizes or aborts by the crash. *Persistent atomicity* [10] requires that the operation linearizes or aborts before any subsequent invocation by the pending thread on any object. *Recoverable linearizability* [2] requires that the operation linearizes or aborts before any subsequent linearization by the pending thread on that same object; under this condition a thread may have more than one operation pending at a time. O_2 demonstrates the non-locality of persistent atomicity; T_2 demonstrates a program order inversion under recoverable linearizability.

have designed data structures that tolerate power failures (e.g. [25,27]), but the semantics of these structures are typically specified informally; the criteria according to which they are correct remain unclear. Guerraoui and Levy have proposed *persistent atomicity* [10] as a safety condition for persistent concurrent objects. This condition ensures that the state of an object will be consistent in the wake of a crash, but it does not provide *locality*: correct histories of separate objects, when merged, will not necessarily yield a correct composite history. Berryhill et al. have proposed an alternative, *recoverable linearizability* [2], which achieves locality but may sacrifice program order after a crash. Earlier work by Aguilera and Frølund proposed *strict linearizability* [1], which preserves both locality and program order but provably precludes the implementation of some wait-free objects for certain (limited) machine models. These safety conditions are illustrated in Fig. 1.

Interestingly, both the lack of locality in persistent atomicity and the loss of program order in recoverable linearizability stem from the assumption that an individual abstract thread may crash, recover, and then continue execution. While well defined, this *failure model* is more general than is normally assumed in real-world systems. More commonly, processes are assumed to fail together, as part of a "full system" crash. A data structure that survives such a crash may safely assume that subsequent accesses will be performed by different threads. We observe that if we consider only full-system crashes (an assumption modeled as a well-formedness constraint on histories), then persistent atomicity and recoverable linearizability are indistinguishable (and thus local). They are also satisfied by existing persistent structures. We use the term *durable linearizability* to refer to this merged safety condition under the restricted failure model.

Independent of failure model, existing theoretical work typically requires that operations become persistent before they return to their caller. In practice, this requirement is likely to impose unacceptable overhead, since persistent memory, while dramatically faster than disk or flash storage, still incurs latencies of hundreds of cycles. To address the latency problem, we introduce *buffered durable linearizability*, which requires only that an operation be "persistently ordered" before it returns. State in the wake of a crash is still required to be consistent, but it need not necessarily be fully up-to-date. Data structures designed with buffering in mind will typically provide an explicit sync method that guarantees, upon its return, that all previously ordered operations have reached persistent memory; an application thread might invoke this method before performing I/O. Unlike its unbuffered variant, buffered durable linearizability is not local: a history may fail to be buffered durably linearizable even if all of its object subhistories are. If the buffering mechanism is shared across all objects, however, an implementation can ensure that all *realizable* histories—those that actually emerge from the implementation—will indeed be buffered durably linearizable: the post-crash states of all objects will be mutually consistent.

At the implementation level, prior work has explored the *memory persistency model* (analogous to a traditional *consistency* model) that governs instructions used to push the contents of cache to NVM. Existing persistency models assume that hardware will track dependencies and automatically write dirty cache lines back to NVM as necessary [8,19,23]. Unfortunately, real-world ISAs require the programmer to request writes-back explicitly. Furthermore, existing persistency models have been explored only for sequentially consistent (SC) [23] or total-store order (TSO) machines [8,19]. At the same time, recent persistency models [19,23] envision functionality not yet supported by commercial ISAs—namely, hardware buffering in an ordered queue of writes-back to persistent memory, allowing *persistence fence* (pfence) ordering instructions to complete without waiting for confirmation from the physical memory device. To accommodate anticipated hardware, we introduce a memory persistency model, *explicit epoch persistency*, that is both buffered and fully relaxed (release consistent).

Just as traditional concurrent objects require not only safety but liveness, so too should persistent objects. We define two optional liveness conditions: First, an object designed for buffered durable linearizability may provide *nonblocking sync*, ensuring that calls to sync complete without blocking. Second, a nonblocking object may provide *bounded completion*, limiting the amount of work done after a crash prior to the completion (if any) of operations interrupted by the crash. As a liveness constraint, bounded completion contrasts with prior art which imposes safety constraints [1,2,10] on completion (see Fig. 1).

We also present a simple transform that takes a data-race-free program designed for release consistency and generates an equivalent program in which the state persisted at a crash is guaranteed to represent a consistent cut across the happens-before order of the original program. When the original program comprises the implementation of a linearizable nonblocking concurrent object, extensions to this transform result in a buffered durably or durably linearizable

object. (If the original program is blocking, additional machinery—e.g., undo logging—may be required. While we do not consider such machinery here, we note that it still requires consistency as a foundation.)

To enable reasoning about our correctness conditions, we extend the notion of linearization points into persistent memory objects, and demonstrate how such *persist points* can be used to argue a given implementation is correct. We also consider optimizations (e.g. elimination) that may safely be excluded from persistence in order to improve performance. Proofs for our lemmas and theorems can be found in an associated technical report [18].

2 Abstract Models

An *abstract history* is a sequence of *events*, which can be: (i) invocations of an object method, (ii) responses associated with invocations, and (iii) system-wide crashes. We use $O.\mathrm{inv}\langle m\rangle_t(params)$ to denote the invocation of operation m on object O, performed by thread t with parameters *params*. Similarly, $O.\mathrm{res}\langle m\rangle_t(retvals)$ denotes the response of m on O, again performed by t, returning *retvals*. A crash is denoted by C.

Given a history $\mathcal{H}$, we use $\mathcal{H}[t]$ to denote the subhistory of $\mathcal{H}$ containing all and only the events performed by thread t. Similarly, $\mathcal{H}[O]$ denotes the subhistory containing all and only the events performed on object O, plus crash events. We use C_i to denote the i-th crash event, and $\mathrm{ops}(\mathcal{H})$ to denote the subhistory containing all events other than crashes. The crash events partition a history as $\mathcal{H} = \mathcal{E}_0\,\mathrm{C}_1\,\mathcal{E}_1\,\mathrm{C}_2\ldots\mathcal{E}_{c-1}\,\mathrm{C}_c\,\mathcal{E}_c$, where c is the number of crash events in $\mathcal{H}$. Note that $\mathrm{ops}(\mathcal{E}_i) = \mathcal{E}_i$ for all $0 \le i \le c$. We call the subhistory $\mathcal{E}_i$ the i-th *era* of $\mathcal{H}$.

Given a history $\mathcal{H} = \mathcal{H}_1\,A\,\mathcal{H}_2\,B\,\mathcal{H}_3$, where A and B are events, we say that A *precedes* B (resp. B *succeeds* A). For any invocation $I = O.\mathrm{inv}\langle m\rangle_t(params)$ in $\mathcal{H}$, the first $R = O.\mathrm{res}\langle m\rangle_t(retvals)$ (if any) that succeeds I in $\mathcal{H}$ is called a *matching response*. A history $\mathcal{S}$ is *sequential* if $\mathcal{S} = I_0\,R_0\ldots I_x\,R_x$ or $\mathcal{S} = I_0\,R_0\ldots I_x\,R_x\,I_{x+1}$, for $x \ge 0$, and $\forall\, 0 \le i \le x, R_i$ is a matching response for I_i.

Definition 1 (Abstract Well-Formedness). *An abstract history $\mathcal{H}$ is said to be* well formed *if and only if $\mathcal{H}[t]$ is sequential for every thread t.*

Note that sequential histories contain no crash events, so the events of a given thread are confined to a single era. (In practice, thread IDs may be re-used as soon as operations of the previous era have completed. In particular, an object with *bounded completion* [Sect. 3.3, Definition 10] can rapidly reuse IDs.)

We consider only well-formed abstract histories. A *completed operation* in $\mathcal{H}$ is any pair (I, R) of invocation I and matching response R. A *pending operation* in $\mathcal{H}$ is any pair $(I, \bot)$ where I has no matching response in $\mathcal{H}$. In this case, I is called a *pending invocation* in $\mathcal{H}$, and any response R such that (I, R) is a completed operation in $\mathcal{H}\,R$ is called a *completing response* for $\mathcal{H}$.

Definition 2 (Abstract Happens-Before). *In any (well-formed) abstract history $\mathcal{H}$ containing events E_1 and E_2, we say that E_1* happens before *E_2*

(denoted $E_1 \prec E_2$) if E_1 precedes E_2 in $\mathcal{H}$ and (1) E_1 is a crash, (2) E_2 is a crash, (3) E_1 is a response and E_2 is an invocation, or (4) there exists an event $\hat{E}$ such that $E_1 \prec \hat{E} \prec E_2$. We extend the order to operations: $(I_1, R_1) \prec (I_2, x)$ if and only if $R_1 \prec I_2$.

Two histories $\mathcal{H}$ and $\mathcal{H}'$ are said to be *equivalent* if $\mathcal{H}[t] = \mathcal{H}'[t]$ for every thread t. We use $\text{compl}(\mathcal{H})$ to denote the set of histories that can be generated from $\mathcal{H}$ by appending completing responses, and $\text{trunc}(\mathcal{H})$ to denote the set of histories that can be generated from $\mathcal{H}$ by removing pending invocations. As is standard, a history $\mathcal{H}$ is *linearizable* if it is well formed, it has no crash events, and there exists some history $\mathcal{H}' \in \text{trunc}(\text{compl}(\mathcal{H}))$ and some legal sequential history $\mathcal{S}$ equivalent to $\mathcal{H}'$ such that $\forall E_1, E_2 \in \mathcal{H}'\ [E_1 \prec_{\mathcal{H}'} E_2 \Rightarrow E_1 \prec_{\mathcal{S}} E_2]$.

Definition 3 (Durable Linearizability). *An abstract history $\mathcal{H}$ is said to be* durably linearizable *if it is well formed and* $\text{ops}(\mathcal{H})$ *is linearizable.*

Durable linearizability captures the idea that operations become persistent before they return; that is, if a crash happens, all previously completed operations remain completed, with their effects visible. Operations that have not completed as of a crash may or may not be completed in some subsequent era. Intuitively, their effects may be visible simply because they "executed far enough" prior to the crash (despite the lack of a response), or because threads in subsequent eras finished their execution for them (for instance, by scanning an "announcement array" in the style of universal constructions [15]). While this approach is simple, it preserves important properties from linearizability, namely *locality* (composability) and *nonblocking progress* [18].

Given a history $\mathcal{H}$ and any transitive order $<$ on events of $\mathcal{H}$, a *$<$-consistent cut* of $\mathcal{H}$ is a subhistory $\mathcal{P}$ of $\mathcal{H}$ where if $E \in \mathcal{P}$ and $E' < E$ in $\mathcal{H}$, then $E' \in \mathcal{P}$ and $E' < E$ in $\mathcal{P}$. In abstract histories, we are often interested in cuts consistent with $\prec$, the happens-before order on events.

Definition 4 (Buffered Durable Linearizability). *A history $\mathcal{H}$ with c crash events is said to be* buffered durably linearizable *if it is well formed and there exist subhistories $\mathcal{P}_0, \ldots, \mathcal{P}_{c-1}$ such that for all $0 \leq i \leq c$, $\mathcal{P}_i$ is a $\prec$-consistent cut of $\mathcal{E}_i$, and $\mathcal{P} = \mathcal{P}_0 \ldots \mathcal{P}_{i-1}\, \mathcal{E}_i$ is linearizable.*

The intent here is that events in the portion of $\mathcal{E}_i$ after $\mathcal{P}_i$ were buffered but failed to persist before the crash. Note that since $\mathcal{P}_i = \mathcal{E}_i$ is a valid $\prec$-consistent cut for all $0 \leq i < c$, we can have $\mathcal{P} = \text{ops}(\mathcal{H})$, and therefore any durably linearizable history is buffered durably linearizable. Note also that buffered durable linearizability is not in general local: if an operation does not persist before it returns, we will not in general be able to ensure that it persists before any operation that follows it in happens-before order unless we arrange for the implementations of separate objects to cooperate.

3 Concrete Models

Concurrent objects are typically implemented by code in some computer language. We want to know if this code is correct. Following standard practice, we

model implementation behavior as a set of *concrete histories*, generated under some language and machine model assumed to be specified elsewhere. Each concrete history consists of a sequence of events, including not only operation invocations, responses, and crash events, but also load, store, and read-modify-write (RMW—e.g., compare-and-swap [CAS]) events, which access the representation of the object. Let $x.\mathsf{ld}_t(v)$ denote a load of variable x by thread t, returning the value v. Let $x.\mathsf{st}_t(v)$ denote a store of v to x by t.

Given a concrete history $\mathcal{H}$, the *abstract history of* $\mathcal{H}$, denoted abstract($\mathcal{H}$), is obtained by eliding all events other than invocations, responses, and crashes. As in abstract histories, we use $\mathcal{H}[t]$ and $\mathcal{H}[O]$ to denote the thread and object subhistories of $\mathcal{H}$. The concept of *era* from Sect. 2 applies verbatim. We say that an event E *lies between* events A and B in a concrete or abstract history $\mathcal{H}$ if A precedes E and E precedes B in $\mathcal{H}$.

Definition 5 (Concrete Well-Formedness). *A concrete history* $\mathcal{H}$ *is* well-formed *if and only if*

1. abstract($\mathcal{H}$) *is well-formed.*
2. *In each thread subhistory of* $\mathcal{H}$*, each memory event either (a) lies between some invocation and its matching response; (b) lies between a pending invocation* I *and the first crash that succeeds* I *in* $\mathcal{H}$ *(if such a crash exists); (c) succeeds a pending invocation* I *if no crash succeeds* I *in* $\mathcal{H}$*.*
3. *The values returned by the loads and RMWs respect the* reads-see-writes *relation (Definition 7, below).*

3.1 Basic Memory Model

For the sake of generality, we build our reads-see-writes relation on the highly relaxed *release consistency* memory model [9]. We allow certain loads to be labeled as *load-acquire* (ld_acq) events and certain stores to be labeled as *store-release* (st_rel) events. We treat RMW events as atomic $\langle \mathsf{ld\_acq}, \mathsf{st\_rel} \rangle$ pairs.

Definition 6 (Concrete Happens-Before). *Given events* E_1 *and* E_2 *of concrete history* $\mathcal{H}$*, we say that* E_1 *is* sequenced-before E_2 *if* E_1 *precedes* E_2 *in* $\mathcal{H}[t]$ *for some thread* t *and (a)* E_1 *is a* ld_acq*, (b)* E_2 *is a* st_rel*, or (c)* E_1 *and* E_2 *access the same location. We say that* E_1 synchronizes-with E_2 *if* $E_2 = x.\mathsf{ld\_acq}_{t'}(v)$ *and* E_1 *is the closest preceding* $x.\mathsf{st\_rel}_t(v)$ *in history order. The* happens-before *partial order on events in* $\mathcal{H}$ *is the transitive closure of sequenced-before order with synchronizes-with order. As in abstract histories, we write* $E_1 \prec E_2$*.*

Note that the definitions of happens-before are different for concrete and abstract histories; which one is meant in a given case should be clear from context.

The release-consistent model corresponds closely to that of the ARM v8 instruction set and can be considered a generalization of Intel's x86 instruction set. Given concrete happens-before, we can define the reads-see-writes relation:

Definition 7 (Reads-See-Writes). *A concrete history $\mathcal{H}$ respects the* reads-see-writes relation *if for each load $R \in \{x.\mathsf{ld}_t(v), x.\mathsf{ld\_acq}_t(v)\}$, there exists a store $W \in \{x.\mathsf{st}_u(v), x.\mathsf{st\_rel}_u(v)\}$ such that either (1) $W \prec R$ and there exists no store W' of x such that $W \prec W' \prec R$ or (2) W is unordered with respect to R under happens-before.*

For simplicity of exposition, we consider the initial value of each variable to have been specified by a store that happens before all other instructions in the history. We consider only well-formed concrete histories in this paper. If case (2) in Definition 7 never occurs in a history $\mathcal{H}$, we say that $\mathcal{H}$ is *data-race-free*.

3.2 Extensions for Persistence

The semantics of instructions controlling the ordering and timing under which cached values are pushed to persistent memory comprise a memory *persistency model* [23]. Since any machine with bounded caches must sometimes evict and write back a line without program intervention, the principal challenge for designers of persistent objects is to ensure that a newer write does not persist before an older write (to some other location) when correctness after a crash requires the locations to be mutually consistent.

Under the *epoch persistency* model of Condit et al. [8] and Pelley et al. [23], writes-back to persistent memory (*persist* operations) are *implicit*—they do not appear in the program's instruction stream. When ordering is required, a program can issue a special instruction (which we call a pfence) to force all of its earlier writes to persist before any subsequent writes. Periods between pfences in a given thread are known as *epochs*. As noted by Pelley et al. [23], it is possible for writes-back to be *buffered*. When necessary, a separate instruction (which we call psync) can be used to wait until the buffer has drained.

Unfortunately, implicit write-back of persistent memory is difficult to implement in real hardware [8,19,23]. Instead, manufacturers have introduced explicit *persistent write-back* (pwb) instructions. These are typically implemented in an eager fashion: a pwb starts the write-back process; a psync waits for the completion of all prior pwbs (under some appropriate definition of "prior").

We generalize proposed implicit persistency models [8,19,23] and real world (explicit) persistency ISAs to define our own, new model, which we call *explicit epoch persistency*. Like real-world explicit ISAs, our persistency model requires programmers to use a pwb to force back data into persistence. Like other buffered models, we provide pfence, which ensures that all previous pwbs are ordered with respect to any subsequent pwbs, and psync, which waits until all previous pwbs have actually reached persistent memory. We assume that persists to a given location respect coherence: the programmer need never worry that a newly persisted value will later be overwritten by the write-back of some earlier value. Unlike prior art, which assumes sequential consistency [23] or total store order [8, 19,20], we integrate our instructions into a relaxed (release consistent) model.

Returning to concrete histories, we use $x.\mathsf{pwb}_t$ to denote a pwb of variable x by thread t, pfence_t to denote a pfence by thread t, and psync_t to denote a

psync by thread t. We amend our definition of concrete histories to include these *persistence events*. We refer to any non-crash event of a concrete history as an *instruction*.

Definition 8 (Persist Ordering). *Given events E_1 and E_2 of concrete history $\mathcal{H}$, with E_1 preceding E_2 in the same thread subhistory, we say that E_1* is persist-ordered before *E_2, denoted $E_1 \lessdot E_2$, if*

(a) $E_1 = \mathsf{pwb}$ and $E_2 \in \{\mathsf{pfence}, \mathsf{psync}\}$;
(b) $E_1 \in \{\mathsf{pfence}, \mathsf{psync}\}$ and $E_2 \in \{\mathsf{pwb}, \mathsf{st}, \mathsf{st\_rel}\}$;
(c) $E_1, E_2 \in \{\mathsf{st}, \mathsf{st\_rel}, \mathsf{pwb}\}$, and E_1 and E_2 access the same location;
(d) $E_1 \in \{\mathsf{ld}, \mathsf{ld\_acq}\}$, $E_2 = \mathsf{pwb}$, and E_1 and E_2 access the same location; or
(e) $E_1 = \mathsf{ld\_acq}$ and $E_2 \in \{\mathsf{pfence}, \mathsf{psync}\}$.

Finally, across threads, $E_1 \lessdot E_2$ if

(f) $E_1 = \mathsf{st\_rel}$, $E_2 = \mathsf{ld\_acq}$, and E_1 synchronizes with E_2.

To identify the values available after a crash, we extend the syntax of concrete histories to allow store events to be labeled as "persisted," meaning that they will be available in subsequent eras if not overwritten. Persisted store labels introduce additional well-formedness constraints:

Definition 9 (Concrete Well-Formedness [augments Definition 5]**).** *A concrete history $\mathcal{H}$ is* well-formed *if and only if it satisfies the properties of Definition 5 and*

4. *For each variable x, at most one store of x is labeled as persisted in any given era. We say the* $(x, 0)$-persisted store *is the labeled store of x in $\mathcal{E}_0$, if there is one; otherwise it is the initialization store of x. For $i > 0$, we say the (x, i)-persisted store is the labeled store of x in $\mathcal{E}_i$, if there is one; otherwise it is the $(x, i-1)$-persisted store.*
5. *For any (x, i)-persisted store W, there is no store W' of x and* psync *event P such that $W \lessdot W' \lessdot P$.*
6. *For any (x, i)-persisted store W, there is no store W' of x and (y, i)-persisted store S such that $W \lessdot W' \lessdot S$.*

These rules ensure that persisted stores compose a $\lessdot$-consistent cut of $\mathcal{H}$. To allow loads to see persisted values in the wake of a crash, we augment the definition of happens-before to declare that the (x, i)-persisted store happens before all events of era $\mathcal{E}_{i+1}$. Definition 7 then stands as originally written.

3.3 Liveness

With strict linearizability, no operation is left pending in the wake of a crash: either it has completed when execution resumes, or it never will. With persistent atomicity and recoverable linearizability, the time it may take to complete a pending operation m in thread t can be expressed in terms of execution steps in t's reincarnation (see Fig. 1). With durable linearizability, which admits no reincarnated threads, any bound on the time it may take to complete m must depend on other threads.

Definition 10 (Bounded Completion). *An object O has* bounded completion *if, for each concrete history $\mathcal{H}$ of O that ends in a crash with an operation m on O still pending, there exists a positive integer k such that for all realizable extensions $\mathcal{H}'$ of $\mathcal{H}$ in which some era of $\mathcal{H}' \smallsetminus \mathcal{H}$ contains at least k instructions issued by an arbitrary thread, either (1) for all realizable extensions $\mathcal{H}''$ of $\mathcal{H}'$, $\mathcal{H}'' \smallsetminus \mathrm{inv}\langle m\rangle$ is buffered durably linearizable or (2) for all realizable extensions $\mathcal{H}''$ of $\mathcal{H}'$, if there exists a completed operation n with $\mathrm{inv}\langle n\rangle \in \mathcal{H}'' \smallsetminus \mathcal{H}'$, then there exists a sequential history $\mathcal{S}$ equivalent to $\mathcal{H}''$ with $m \prec_{\mathcal{S}} n$.*

Briefly: by k post-crash instructions by any thread, m completes, if it ever will.

It is also desirable to discuss progress towards persistence. Under durable linearizability, every operation persists before it responds, so any liveness property (e.g. lock freedom) that holds for method invocations also holds for persistence. Under buffered durable linearizability, the liveness of persist *ordering* is subsumed in method invocations.

As noted in Sect. 1, data structures for buffered persistence will typically need to provide a sync method that guarantees, upon its return, that all previously ordered operations have reached persistent memory. If sync is not rolled into operations, then buffering (and sync) need to be coordinated across all mutually consistent objects, for the same reason that buffered durable linearizability is not a local property (Sect. 2). The existence of sync impacts the definition of buffered durable linearizability. In Definition 4, all abstract events that precede a sync instruction in their era must appear in $\mathcal{P}$, the sequence of consistent cuts. For a set of nonblocking objects, it is desirable that the shared sync method be wait-free or at least obstruction free—a property we call *nonblocking sync*. (As sync is shared, lock freedom doesn't seem applicable.)

4 Implementations

Given our prior model definitions and correctness conditions, we present an automated transform that takes as input a concurrent multi-object program written for release consistency and transient memory, and turns it into an equivalent program for explicit epoch persistency. Rules (T1) through (T5) of our transform (below) preserve the happens-before ordering of the original concurrent program: in the event of a crash, the values present in persistent memory are guaranteed to represent a $\prec$-consistent cut of the pre-crash history. Additional rules (T6) through (T8) serve to preserve real-time ordering not captured by happens-before but required for durable linearizability. The intuition behind our transform is that, for nonblocking concurrent objects, a cut across the happens-before ordering represents a valid static state of the object [22]. For blocking objects, additional recovery mechanisms (not discussed here) may be needed to move the cut if it interrupts a failure-atomic or critical section [5,7,17,26].

The following rules serve to preserve happens-before ordering into persist-before ordering. Their key observation is that a thread t which issues a $x.\mathsf{st\_rel}_t(v)$ cannot atomically ensure the value's persistence. Thus, the subsequent thread u which synchronizes-with $x.\mathsf{ld\_acq}_u(v)$ shares responsibility for x's persistence.

(T1) Immediately after $x.\mathsf{st}_t(v)$, write back the value by issuing $x.\mathsf{pwb}_t$.
(T2) Immediately before $x.\mathsf{st\_rel}_t(v)$, issue a pfence; immediately afterward, write back the value by issuing $x.\mathsf{pwb}_t$.
(T3) Immediately after $x.\mathsf{ld\_acq}_t(v)$, write back the loaded value by issuing $x.\mathsf{pwb}_t$, then issue a pfence.
(T4) Handle CAS instructions as atomic $\langle x.\mathsf{ld\_acq}_t(v), x.\mathsf{st\_rel}_t(v')\rangle$ pairs: immediately before the pair, issue a pfence; immediately afterward, write back the (potentially modified) value by issuing $x.\mathsf{pwb}_t$, then issue a pfence. (Extensions for other RMW instructions are straightforward.)
(T5) Take no persistence action on loads.

In the wake of a crash, the values present in persistent memory will reflect, by Definition 9, a consistent cut across the (partial) persist ordering ($\lessdot$) of the preceding era. We wish to show that in any program created by our transform, it will also reflect a consistent cut across that era's happens-before ordering ($\prec$). Mirroring condition 6 of concrete well-formedness (Definition 9), but with $\prec$ instead of $\lessdot$, we can prove [18]:

Lemma 1. *Consider a concrete history $\mathcal{H}$ emerging from our transform. For any location x and (x,i)-persisted store $A \in \mathcal{H}$, there exists no store A' of x, location y, and (y,i)-persisted store $B \in \mathcal{H}$ such that $A \prec A' \prec B$.*

Unfortunately, preservation of happens-before is not enough to give us durable linearizability: we also need to preserve the "real-time" order of non-overlapping operations (Definition 2, clause 3) in different threads. (As in conventional linearizability, "real time" serves as a stand-in for forms of causality—e.g., loads and stores of variables outside of operations—that are not captured in our histories.)

For objects that are (non-buffered) durably linearizable, we simply need to ensure that each operation persists before it returns:

(T6) Immediately before $O.\mathrm{res}\langle m\rangle_t$, issue a psync.

For buffered durably linearizable objects, we leave out the psync and instead introduce a shared global variable G:

(T7) Immediately before $O.\mathrm{res}\langle m\rangle_t$, issue a pfence, then issue $G.\mathsf{st\_rel}_t(g)$, for some arbitrary fixed value g.
(T8) Immediately after $O.\mathrm{inv}\langle m\rangle_t$, issue $G.\mathsf{ld\_acq}_t(g)$, for the same fixed value g, then issue a pfence.

To facilitate our proof of correctness [18], we introduce the notion of an *effective history* for $\mathcal{H}$. This history leaves out both the crashes of $\mathcal{H}$ and, in each era, the suffix of each thread's execution that fails to reach persistence before the crash. We can then prove (Lemma 2) that any effective history of a program emerging from our transform is itself a valid history of that program (and could have happened in the absence of crashes), and (Lemma 3) that the (crash-free) abstract history corresponding to the effective history is identical to

some concatenation of $\prec$-consistent cuts of the eras of the (crash-laden) abstract history corresponding to $\mathcal{H}$. These two lemmas then support our main result (Theorem 1).

Definition 11. *Consider a concrete history* $\mathcal{H} = \mathcal{E}_0\,\mathrm{C}_1\,\mathcal{E}_1 \ldots \mathcal{E}_{c-1}\,\mathrm{C}_c\,\mathcal{E}_c$. *For any thread* t *and era* $0 \le i < c$, *let* E_i^t *be the last store in* $\mathcal{E}_i[t]$ *that either is a persisted store or happens before some persisted store in* $\mathcal{E}_i$. *Let* B_i^t *be the last non-store instruction that succeeds* E_i^t *in* $\mathcal{E}_i[t]$, *with no stores by* t *in-between (or, if there is no such instruction,* E_i^t *itself). Finally, for* $0 \le j < c$, *let* $\mathcal{P}_j$ *be the subhistory of* $\mathcal{E}_j$ *obtained by removing all persistence events and, for each* t, *all events that follow* B_j^t *in* $\mathcal{E}_j[t]$. *The* effective concrete history *of* $\mathcal{H}$ *at era* i, *denoted* $\mathrm{effective}_i(\mathcal{H})$, *is the history* $\mathcal{P}_0 \ldots \mathcal{P}_{i-1}\mathcal{E}_i$.

Lemma 2. *Consider a nonblocking, data-race-free program* $\mathbb{P}$, *and the transformed program* $\mathbb{P}'$. *For any realizable concrete history* $\mathcal{H}$ *of* $\mathbb{P}'$, *and any* $0 \le i \le c$, $\mathrm{effective}_i(\mathcal{H})$ *is a realizable concrete history of* $\mathbb{P}$.

Lemma 3. *Consider a nonblocking, data-race-free program* $\mathbb{P}$, *and the transformed program* $\mathbb{P}'$. *For any realizable concrete history* $\mathcal{H}$ *of* $\mathbb{P}'$, *and any* $0 \le i \le c$, *the history* $\mathrm{abstract}(\mathrm{effective}_i(\mathcal{H}))$ *is precisely* $\mathcal{P}_0^a \ldots \mathcal{P}_{i-1}^a\,\mathcal{E}_i^a$, *where* $\mathcal{E}_i^a$ *is the* i*-th era of* $\mathrm{abstract}(\mathcal{H})$, *and* $\mathcal{P}_i^a$ *is a* $\prec$*-consistent cut of* $\mathcal{E}_i^a$.

Theorem 1 (Buffered Durable Linearizability). *If a nonblocking, data-race-free program* $\mathbb{P}$ *is linearizable, the transformed program* $\mathbb{P}'$ *is buffered durably linearizable.*

In addition to the correctness properties of our automated transform, we can characterize other properties of the code it generates. For example, the transformed implementation of a nonblocking concurrent object requires no change to persistent state before relaunching threads—that is, it has a *null recovery procedure*. Moreover, any set of transformed objects will share a wait-free sync method (a single call to psync).

In each operation on a transient linearizable concurrent object, we can identify some instruction within as the operation's *announce point*: once execution reaches the announce point, the operation may linearize without its thread taking additional steps. Wait-free linearizable objects sometimes have announce points that are not atomic with their linearization points. In most nonblocking objects, however, the announce point *is* the linearization point, a property we call *unannounced*. This property results in stronger correctness properties in the persistent version when the object is transformed. The result of transform when applied to an object whose operations are unannounced is strictly linearizable. Perhaps surprisingly, our transform does not guarantee bounded completion, even on wait-free objects. Pending announced operations may be ignored for an arbitrary interval before eventually being *helped* to completion [4] [14, Sect. 4.2.5].

4.1 Persist Points

Linearizability proofs for transient objects are commonly based on the notion of a *linearization point*—an instruction between an operation's invocation and response at which the operation appears to "take effect instantaneously" [16]. In simple objects, linearization points may be statically known. In more complicated cases, one may need to reason retrospectively over a history in order to identify the linearization points, and the linearization point of an operation need not necessarily be an instruction issued by the invoking thread.

The problem for persistent objects is that an operation cannot generally linearize and persist at the same instant. Clearly, it will need to linearize first; otherwise it will not know what values to persist. Unfortunately, as soon as an operation (call it m) linearizes, other operations can see its state, and might, naively, linearize *and persist* before m had a chance to persist. The key to avoiding this problem is for every operation n to ensure that any predecessor on which it depends has persisted (in the unbuffered case) or persist-ordered (with global buffering) before n itself linearizes. To preserve real-time order, n must also persist (or persist-order) before it returns.

Theorem 2 (Persist Points). *Suppose that for each operation m of object O it is possible to identify not only a linearization point l_m between* $\mathrm{inv}\langle m\rangle$ *and* $\mathrm{res}\langle m\rangle$ *but also a* persist point *instruction p_m between l_m and* $\mathrm{res}\langle m\rangle$ *such that (1) "all stores needed to capture m" are written back to persistent memory, and a* pfence *issued, before p_m; and (2) whenever operations m and n overlap, linearization points can be chosen such that either $p_m \lessdot l_n$ or l_n precedes l_m. Then O is (buffered) durably linearizable.*

The notion of "all stores needed to capture m" will depend on the details of O. In simple cases (e.g., those emerging from our automated transform), those stores might be all of m's updates to shared memory. In more optimized cases, they might be a proper subset (as discussed below). Generally, a nonblocking persistent object will embody helping: if an operation has linearized but not yet persisted, its successor operation must be prepared to push it to persistence.

4.2 Practical Applications

A variety of standard concurrent data structure techniques can be adapted to work with both durable and strict linearizability and their buffered variants. While our automated transform can be used to create correct persistent objects, judicious use of transient memory can often reduce the overhead of persistence without compromising correctness. For instance, announcement arrays [13] are a common idiom for wait-free helping mechanisms. Implementing a transient announcement array [2] while using our transform on the remainder of the object state will generally provide a (buffered) strictly linearizable persistent object.

Other data structure components may also be moved into transient memory. Elimination arrays [12] might be used on top of a durably or strictly linearizable data structure without compromising its correctness. The flat combining

technique [11] is also amenable to persistence. Combined operations can be built together and ordered to persistence with a single pfence, then linked into the main data structure with another, reducing pfence instructions per operation. A transient combining array will generally result in a strictly linearizable object; leaving it persistent memory results in a durably linearizable object.

Several library and run-time systems have already been designed to take advantage of NVM; many of these can be categorized by the presented correctness conditions. Strictly linearizable examples include trees [25,27], file systems [8], and hash maps [24]. Buffered strictly linearizable data structures also exist [21], and some libraries explicitly enable their construction [3,5]. Durably (but not strictly) linearizable data structures are a comparatively recent innovation [17].

5 Conclusion

This paper has presented a framework for reasoning about the correctness of persistent data structures, based on two key assumptions: *full-system crashes* at the level of abstract histories and *explicit write-back and buffering* at the level of concrete histories. For the former, we capture safety as (buffered) *durable linearizability*; for the latter, we capture anticipated real-world hardware with *explicit epoch consistency*, and observe that both buffering and persistence introduce new issues of liveness. Finally, we have presented both an automatic mechanism to transform a transient concurrent object into a correct equivalent object for explicit epoch persistency and a notion of *persist points* to facilitate reasoning for other, more optimized, persistent objects.

References

1. Aguilera, M.K., Frølund, S.: Strict linearizability and the power of aborting. Technical report. HPL-2003-241, HP Labs (2003)
2. Berryhill, R., Golab, W., Tripunitara, M.: Robust shared objects for non-volatile main memory. In: International Conference on Principles of Distributed Systems, Rennes, France (2015)
3. Boehm, H.J., Chakrabarti, D.: Persistence programming models for non-volatile memory. Technical report. HP-2015-59, HP Laboratories (2015)
4. Censor-Hillel, K., Petrank, E., Timnat, S.: Help! In: ACM Symposium on Principles of Distributed Computing, Donostia-San Sebastián, Spain (2015)
5. Chakrabarti, D.R., Boehm, H.J., Bhandari, K.: Atlas: leveraging locks for non-volatile memory consistency. In: 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, Portland, OR (2014)
6. Chatzistergiou, A., Cintra, M., Viglas, S.D.: Rewind: recovery write-ahead system for in-memory non-volatile data-structures. Proc. VLDB Endow. **8**(5), 497–508 (2015)
7. Coburn, J., Caulfield, A.M., Akel, A., Grupp, L.M., Gupta, R.K., Jhala, R., Swanson, S.: NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In: 16th International Conference on Architectural Support for Programming Languages and Operating Systems, Newport Beach, CA (2011)

8. Condit, J., Nightingale, E.B., Frost, C., Ipek, E., Lee, B., Burger, D., Coetzee, D.: Better I/O through byte-addressable, persistent memory. In: 22nd ACM Symposium on Operating Systems Principles, Big Sky, MT (2009)
9. Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.: Memory consistency and event ordering in scalable shared-memory multiprocessors. In: 17th International Symposium on Computer Architecture, Seattle, WA (1990)
10. Guerraoui, R., Levy, R.: Robust emulations of shared memory in a crash-recovery model. In: 24th International Conference on Distributed Computing Systems, Santa Fe, NM (2004)
11. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: 22nd ACM Symposium on Parallelism in Algorithms and Architectures, Santorini, Greece (2010)
12. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: 16th ACM Symposium on Parallelism in Algorithms and Architectures, Barcelona, Spain (2004)
13. Herlihy, M.: A methodology for implementing highly concurrent data objects. ACM Trans. Program. Lang. Syst. **15**(5), 745–70 (1993)
14. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, San Francisco (2008)
15. Herlihy, M.P.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1), 124–49 (1991)
16. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–92 (1990)
17. Izraelevitz, J., Kelly, T., Kolli, A.: Failure-atomic persistent memory updates via JUSTDO logging. In: 21st International Conference on Architectural Support for Programming Languages and Operating Systems, Atlanta, GA (2016)
18. Izraelevitz, J., Mendes, H., Scott, M.L.: Linearizability of persistent memory objects under a full-system-crash failure model. Technical report. 999, Dept. of Computer Science, Univ. of Rochester (2016)
19. Joshi, A., Nagarajan, V., Cintra, M., Viglas, S.: Efficient persist barriers for multicores. In: 48th International Symposium on Microarchitecture, Waikiki, HI (2015)
20. Kolli, A., Pelley, S., Saidi, A., Chen, P.M., Wenisch, T.F.: High-performance transactions for persistent memories. In: 21st International Conference on Architectural Support for Programming Languages and Operating Systems, Atlanta, GA (2016)
21. Moraru, I., Andersen, D.G., Kaminsky, M., Tolia, N., Binkert, N., Ranganathan, P.: Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In: ACM Conference on Timely Results in Operating Systems, Farmington, PA (2013)
22. Nawab, F., Chakrabarti, D.R., Kelly, T., Morrey III., C.B.: Procrastination beats prevention: timely sufficient persistence for efficient crash resilience. In: 18th International Conference on Extending Database Technology, Brussels, Belgium (2015)
23. Pelley, S., Chen, P.M., Wenisch, T.F.: Memory persistency. In: 41st International Symposium on Computer Architecture, Minneapolis, MN (2014)
24. Schwalb, D., Dreseler, M., Uflacker, M., Plattner, H.: NVC-hashmap: a persistent and concurrent hashmap for non-volatile memories. In: 3rd VLDB Workshop on In-Memory Data Management and Analytics, Kohala, HI (2015)
25. Venkataraman, S., Tolia, N., Ranganathan, P., Campbell, R.H.: Consistent and durable data structures for non-volatile byte-addressable memory. In: 9th USENIX Conference on File and Storage Technologies, San Jose, CA (2011)

26. Volos, H., Tack, A.J., Swift, M.M.: Mnemosyne: lightweight persistent memory. In: 16th International Conference on Architectural Support for Programming Languages and Operating Systems, Newport Beach, CA (2011)
27. Yang, J., Wei, Q., Chen, C., Wang, C., Yong, K.L., He, B.: NV-Tree: reducing consistency cost for NVM-based single level systems. In: 13th USENIX Conference on File and Storage Technologies, Santa Clara, CA (2015)