Spirea: A Mechanized Concurrent Separation Logic for Weak Persistent Memory

SIMON FRIIS VINDUM, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

Weak persistent memory (a.k.a. non-volatile memory) is an emerging technology that offers fast byte-addressable durable main memory. A wealth of algorithms and libraries has been developed to explore this exciting technology. As noted by others, this has led to a significant verification gap. Towards closing this gap, we present Spirea, the first concurrent separation logic for verification of programs under a weak persistent memory model. Spirea is based on the Iris and Perennial verification frameworks, and by combining features from these logics with novel techniques it supports high-level modular reasoning about crash-safe and thread-safe programs and libraries. Spirea is fully mechanized in the Coq proof assistant and allows for interactive development of proofs with the Iris Proof Mode. We use Spirea to verify several challenging examples with modular specifications. We show how our logic can verify thread-safety and crash-safety of non-blocking durable data structures with null-recovery, in particular the Treiber stack and the Michael-Scott queue adapted to persistent memory. This is the first time durable data structures have been verified with a program logic.

CCS Concepts: • **Theory of computation → Separation logic**; **Logic and verification**.

Additional Key Words and Phrases: weak memory, persistent memory, non-volatile memory, persistency, program verification, program logic, Iris

ACM Reference Format:
Simon Friis Vindum and Lars Birkedal. 2023. Spirea: A Mechanized Concurrent Separation Logic for Weak Persistent Memory. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 244 (October 2023), 37 pages. https://doi.org/10.1145/3622820

1 INTRODUCTION

In the traditional storage hierarchy programmers can choose between fast, but volatile, main memory and non-volatile, but slower, secondary storage. Persistent memory (a.k.a. non-volatile memory) is an exciting emerging technology that, uniquely, offers both fast random access at byte granularity and persistence of data in the absence of power and across system crashes. It thus shakes up the traditional storage hierarchy with a new abstraction: storage that is suitable both as main memory and as durable storage of data.

A wealth of algorithms, libraries, and tools have been developed for persistent memory, exploring the new potential. This includes durable data structures [Cai et al. 2021; Friedman et al. 2018], memory allocators [Schwalb et al. 2015], garbage collectors [Cai et al. 2020], transactions [Ramalhete et al. 2021; Volos et al. 2011], key-value stores [Chen et al. 2020; Kaiyrakhmet et al. 2019], and language-level support for persistent memory [George et al. 2020], just to mention a few. An important class of data structures that is new and unique to persistent memory is durable data-structures with *null-recovery* [Izraelevitz et al. 2016]. These reside in persistent memory and are preserved across crashes with *no recovery being needed* after a crash to maintain their consistency.

Ensuring correctness when programming for persistent memory is, however, extremely challenging. Since data stored in persistent memory is expected to be permanent, programs for persistent memory must be *crash-safe*. Thus, programmers must ensure that if the system crashes (which

Authors' addresses: Simon Friis Vindum, vindum@cs.au.dk, Department of Computer Science, Aarhus University, Denmark; Lars Birkedal, Department of Computer Science, Aarhus University, Denmark, birkedal@cs.au.dk.

2023. 2475-1421/2023/10-ART244
https://doi.org/10.1145/3622820

can happen non-deterministically at any time, *e.g.*, due to power failure) then, after the crash, the content of the persistent memory should be in a consistent state from which recovery is possible.

Moreover, due to the volatile caches on contemporary CPUs, writes to persistent memory are buffered. They occur *asynchronously* and may reach persistent memory in a different order than the one in which they were carried out. This *persistent memory order* (or persist order) does not coincide with the *weak memory order*, the order in which the CPU guarantees that writes by one thread are made visible to other threads. Hence, a program can be correct for weak memory (by taking into account the weak memory order), but not correct for persistent memory (by failing to take the persistent memory order properly into account). To tame this non-determinism, modern instruction sets such as x86 and ARM offer various flush and fence instructions, which programmers can insert between writes to enforce a desired persist order. These instructions are expensive, though, and should only be used when necessary.

One solution to ensure correctness in the presence of these challenges is, of course, to formally verify programs for persistent memory using a program logic. However, as Raad et al. [2020a] identified, there is a significant *verification gap*: The development of algorithms and libraries for persistent memory is far ahead of formal verification techniques for persistent memory. As a first step towards closing this gap two program logics have been developed: Persistent Owicki-Gries (POG) [Raad et al. 2020a] and Pierogi [Bila et al. 2022]. Both are adaptations of the Owicki-Gries proof system and for reasoning about programs under the machine-level x86-TSO memory model. However, since these logics are based on Owicki-Gries they only support a very simple first-order sequential programming language and do not include features such as separation, (user defined) ghost state, higher-order reasoning, and abstract specifications. This results in a lack of modularity that is evident, for instance, in [Raad et al. 2020a], where to verify an example using a lock, the lock and the client of the lock are verified together using a global invariant with knowledge about the internals of both. It is not possible to give the lock an abstract specification, verify it in isolation, and reuse the specification with multiple clients. In contrast, modern concurrent separation logics (CSLs), such as Iris [Jung et al. 2018], scale to much richer programming languages and support the aforementioned features. We thus think that the next step to closing the verification gap is to develop a CSL for persistent memory, and that is exactly what we do in this paper.

1.1 Challenges

Prior work has explored the application of CSL to weak memory and to persistency *individually*. The RSL and GPS logics has spawned a line of logics for weak (but not persistent) memory [Doko and Vafeiadis 2016; Kaiser et al. 2017; Turon et al. 2014; Vafeiadis and Narayan 2013]. The Perennial logic, which is a state-of-the-art CSL for reasoning about crash-safety, and its predecessor Crash Hoare Logic applies to programs that use durable secondary storage (but without any weak behaviors) [Chajed 2022; Chajed et al. 2019, 2021; Chen et al. 2016]. These logics have been successful in their respective domains but no CSL has been developed for the *weak persistency* found in persistent memory. As persistent memory combines challenging aspects from both weak memory and persistency a natural approach is to learn from the above-mentioned logics and try to adapt their techniques into a logic for persistent memory. As it turns out, there are however serious obstacles to such an endeavor:

Non-deterministic crashes. In a strong persistency model, such as the one considered for Crash Hoare Logic and Perennial, crashes are *deterministic*. This means that if a crash occurs at a given program point the state of the machine after the crash is uniquely determined by its state before the crash at that program point. The durable storage is completely unaffected by the crash whereas the content of volatile memory is entirely lost. At the program logic level this means that some logical resources are kept unchanged at a crash while others are discarded. Perennial includes a *post-crash* modality, $\langle \mathsf{PC} \rangle$, that carries out this transformation. All rules for their post-crash modality have the form $R \vdash \langle \mathsf{PC} \rangle\, R$, which means that the resource R is preseved during a crash. If a resource P is lost at a crash this is simply encoded by having no such rule for P.

For persistent memory the persistency model is weak due to the asynchronous nature of writes and fences. This means that the crash step is *non-deterministic*. As such, resources are not merely kept or lost at a crash; instead they are non-deterministically kept, discarded, or *changed*. Hence, the straightforward behavior of Perennial's post-crash modality is no longer sufficient and its model, which relies on changing ghost names for lost resources, is not applicable either! We thus introduce a more sophisticated post-crash modality and prove it sound using a more subtle model.

Sound invariants. It is well-known that Iris-style invariants are unsound for weak memory. To overcome this, CSLs for weak memory have had to restrict invariants in various ways. One approach taken by GPS, iGPS and iRC11 is to associate invariants with specific locations and only allow access to their content when physically synchronizing with the location. We observe that in a persistent memory setting even these restricted invariants allows for resource transfer that is unsound for persistent memory. In particular, in weak memory if a RMW (read-modify-write) operation is successful then the overwritten value can never be read again by another RMW operation. The weak memory invariants rely on this property for certain types of resource transfer. But, in persistent memory, a write made by an RMW operation might be lost at a crash, and the overwritten value will then be observable again after the crash.

Additionally, we want invariants that are strong enough to handle durable data structures with null-recovery. The obvious way to encode at the logic level that a data-structure is preserved across crashed is to say that its invariant (inside its representation predicate) is preserved under the post-crash modality. However, it is not clear how an Iris invariant can soundly interact with a post-crash modality. Indeed, in Perennial, which uses Iris invariants, one cannot use the post-crash modality to establish that an invariant holds after a crash. Instead, Perennial relies on recovery code to establish *new* invariants after a crash, but this approach does not work for null-recovery where there is no recovery code.

A somewhat subtle point is that the issues with reconciling Iris invariants and crashes also pose challenges regarding modeling of the logic. Prior Iris-based logics for weak memory use Iris invariants internally to model their more restrictive user-level invariants. But if invariants can not survive crashes, then they can not be used in the model either.

Persistent memory instructions. Persistent memory models usually involve some combination of flushes and fences to restrict the persist order when necessary. These instructions are specific to persistent memory and are not addressed by prior separation logics. We consider a weak flush instruction that may be reordered with respect to other instructions up to a fence. As noted by Raad et al. [2020a] such a flush instruction is difficult to reason about as its effect does not take place at the program point of the flush. As for fences we consider both asynchronous fences and synchronous fences.

1.2 Our Contributions

This paper contributes Spirea, the first CSL for weak persistency in general and persistent memory in particular. We use the *explicit epoch persistency* model by Izraelevitz et al. [2016].[1] This model is a slight generalization of the x86 and the ARM persistency models which can be efficiently implemented on both processors. As the model is slightly weaker than x86 and ARM, programs that

[1]Not to be confused with the (implicit) epoch persistency model which cannot be efficiently implemented on x86 or ARM.

are proven correct for this model are correct for both x86 and ARM. Similarly, reasoning principles that apply for this model are more general and are sound also for x86 and ARM. As such, the ideas in Spirea are generally applicable and can also be used, for instance, in logics specifically for x86 and ARM. In §2 we give an intuitive account of the persistency model as well as the consistency model and explain the verification challenges in more detail. Izraelevitz et al. [2016] define the explicit epoch persistency model in a declarative style, as a number of ordering constraints on abstract histories. Such a formulation is not well-suited for reasoning in a CSL, so we recast their model as a view-based small-step operational semantics (see §5.1) that can be used with the Perennial and Iris logical frameworks. As our focus in this paper is squarely on the logic we do not establish a formal correspondence between Izraelevitz et al.'s formulation and ours but instead leave this to future work.

Our logic improves the state-of-the-art both in terms the programming language features it supports, the expressivity and power of the logic, and in the scope of the case studies we have verified. Our programming language λ_{pmem} includes many features that are not supported by the Owicki-Gries based logics, most importantly: dynamic allocation of references, dynamic forking of threads, functions (including higher-order recursive functions and closures), and compound data types. As for the logic, Spirea is a higher-order separation logic and includes all the usual features in Iris based separation logics (except for those that are unsound in our setting). For reasoning about crashes Spirea contains features equivalent to those of Perennial. We cover this background in §3.

To tackle the above-mentioned challenges, Spirea includes the following key innovations:

(1) A *resource changing posts crash modality* that can account for the non-deterministic changes in resources at crashes under weak persistency. Our post-crash modality supports rules of the form $R \vdash \langle \mathsf{PC} \rangle\, R'$, where R' reflects how R is non-deterministically affected by the crash. We make this possible by modelling our post-crash modality using an *exchange resource*. This can be seen as a generalization of the model of Perennial's post-crash modality: the Perennial model is the special case where the exchange resource is the empty resource.
(2) *Crash-aware invariants*, which, in contrast to Iris-style and GPS-style invariants, are sound under weak persistency. Soundness of Spirea crash-aware invariants relies on having novel proof rules for transfer of resources in and out of invariants. Our Spirea invariants are *crash-aware*, meaning that they can be preserved under our post-crash modality and thus facilitate resource transfer between code executing before and after a crash. This is the first time a separation logic contains invariants that can be used to this end. We devise a novel model for our invariants that does *not* rely on Iris invariants.
(3) An assortment of features to handle persistent memory instructions: *Post-fence modalities*, a *post-crash flush modality*, and *state lower-bounds* w.r.t. fences. These work in tandem to reason about weak flushes and synchronous and asynchronous fences.

We explain these in depth in §4 where we give a high level introduction to Spirea, explain its design, and present several examples.

Spirea and its high-level reasoning rule are modelled on top of a lower-level logic called BaseSpirea. This logic, in turn, is modelled using an instantiation of the Perennial program logic and using the Iris base logic. In §5 we state the soundness result in terms of the operational semantics. We also give an overview of the semantic model and the proof of soundness to the extent that space permits. For the full details regarding the model and the soundness proof we refer the reader to our mechanization.

Spirea and all our results are fully mechanized in the Coq proof assistant. The mechanization allows for interactive development of proofs using the Iris proof mode. The development is available

$$
\begin{array}{rcl}
v \in \textsc{Val} & ::= & () \mid i \in \mathbb{Z} \mid \ell \in \mathit{Loc} \mid \mathtt{true} \mid \mathtt{false} \mid (v, v) \mid \mathtt{inj}_1\, v \mid \mathtt{inj}_2\, v \mid \mathtt{rec}\ f(x) = e \mid \cdots \\
e \in \textsc{Exp} & ::= & x \mid v \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \mid (e, e) \mid \pi_1\, e \mid \pi_2\, e \mid \mathtt{inj}_1\, e \mid \mathtt{inj}_2\, e \mid e\ e \mid \cdots \\
& \mid & \mathtt{match}\ e\ \mathtt{with}\ \mathtt{inj}_1\, x \Rightarrow e \mid \mathtt{inj}_2\, x \Rightarrow e \mid \mathtt{fork}\,\{e\} \mid \mathtt{ref}_a\, e \mid !_a\, e \mid e :=_a e \\
& \mid & \mathtt{CAS}\ e\ e\ e \mid \mathtt{FAA}\ e\ e\ e \mid \mathtt{flush}\ e \mid \mathtt{fence} \mid \mathtt{fence_{sync}} \qquad\qquad \text{for } a \in \{\mathrm{na}, \mathrm{at}\}
\end{array}
$$

Fig. 1. The syntax of λ_{pmem}

```
x :=na 37;  ||  if !at y = 1
y :=at 1    ||  then
            ||    assert (!na x = 37)
```

(a) Message passing (MP)

```
x :=na 37;  ||  if !at y = 1           if !at z = 1
flush x;    ||  then               ↻   then
fence;      ||    fence;                 assert (!na x = 37)
y :=at 1    ||    z :=na 1
```

(b) Durable MP

```
x :=na 37;       if !na y = 1
flush x;     ↻   then
fence;             assert (!na x = 37)
y :=na 1
```

(c) Flush and fence

```
            ||  if !at y = 1            if !at z = 1
x :=na 37;  ||  then flush x;       ↻   then
y :=at 1    ||       fence;               assert (!na x = 37)
            ||       z :=na 1
```

(d) Optimized durable MP

Fig. 2. Examples of programs that use weak and persistent memory operations

online at https://github.com/logsem/spirea and as an artifact accompanying this paper ??. We have used the mechanization of our logic to formally verify a range of examples and case studies. We cover a number of these in §6. The case studies demonstrate how our logic is capable of verifying tricky synthetic examples, that it can give modular and compositional specifications to thread-safe and crash-safe libraries, and even verify entire durable data structures with null-recovery. For the latter we have verified crash-safety and thread-safety of both a durable version of the Treiber stack and the Michael-Scott queue. This is the first time durable data structures have been verified with a program logic.

In §7 we discuss related and future work.

2 PERSISTENT MEMORY VERIFICATION CHALLENGES

Before we can introduce Spirea we must first understand the kinds of programs that it aims to verify correctness of and the challenges involved in this. To this end we introduce our programming language λ_{pmem}. Its syntax is seen in Fig. 1. We use highlighted text to indicate the parts of the language that are related to persistent memory only. Loosely speaking, if we erased those parts we would get a language for weak, but not persistent, memory.

λ_{pmem} is a lambda-calculus with standard features (recursive functions, booleans, products, sums, *etc.*), fork-based concurrency, references with dynamic allocation, and operations for weak persistent memory. The expression **fork** $\{e\}$ spawns a new thread that evaluates e in parallel with existing threads. We use the notation $e_1 \parallel e_2$ for the parallel execution of e_1 and e_2, which is derivable from **fork**. We define **assert** to be function that is unsafe (gets stuck) if its argument is not **true**. The language features a weak persistent memory model. The full formal operational semantics appears in Appendix A. In this section we give an intuitive explanation of the memory model illustrated by the examples in Fig. 2. But first we fix some terminology.

A *consistency model* specifies the semantics of shared memory by restricting the *weak memory order*, the order of memory operations across threads. A concurrent program that correctly accounts for interleavings and the weak memory order is *thread-safe*. A *persistency model* specifies the semantics of persistent memory by restricting the *persist order*, the order in which writes may reach the persistent memory[Pelley et al. 2014]. A program using durable storage that correctly accounts for crashes and the persist order is *crash-safe*. The mentioned orders are defined using the *program order*, the order in which memory operations are issued by the program.

2.1 Release-Acquire and Non-Atomic Consistency

We use a highly relaxed consistency model closely resembling the release-acquire and non-atomic fragment of C11.[2] The memory operations for allocations ($\mathtt{ref}_a$), writes ($:=_a$), and reads ($!_a$) are annotated with a *memory access mode* $a \in \{\mathsf{na}, \mathsf{at}\}$. Allocations are considered a form of writes in the memory model. The access modes na and at are *non-atomic* and *atomic* access, respectively.

Non-atomic access is to be used when there are no races on data. For instance, when a thread uses a location exclusively or when synchronization has been established through other means, *e.g.*, through a lock or atomic operations (explained below). Non-atomic writes ($:=_{\mathsf{na}}$) performed by one thread give no guarantees on the order in which other threads may see them. This implies that it would be unsafe to use a non-atomic write to y in the example in Fig. 2a. The right thread might read 1 from y without also reading 37 from x.

To ensure a desired weak memory order across threads, atomic access must be used. An atomic write ($:=_{\mathsf{at}}$) is called a *release-write* and an atomic read ($!_{\mathsf{at}}$) is called an *acquire-read*. If an acquire-read reads a value written by a release-write we say that the acquire-read *synchronizes* with the release-write. In this case, the write is ordered before the read in the weak memory order. Furthermore, a release-write is ordered after all preceding (in program order) memory operations, and an acquire-read is ordered before all succeeding (in program order) reads and writes. Together, this means that when a thread, call it t_1, performs an acquire-read and synchronizes with a release-write of another thread, say t_2, then t_1 becomes "aware of" (or acquires) all the writes that t_2 was aware of at the time of writing. This is exemplified by the message passing (MP) example in Fig. 2a where the use of atomic operations make the assertion safe. When the sender thread writes 1 to y it is aware of the write of 37 to x (since it wrote it itself, program order). Hence, if the receiving thread reads 1 from y it also becomes aware of the write to x, thus the following read of x is certain to yield 37, and the assert will succeed.

The read-modify-write (RMW) operations **CAS** (compare-and-set) and **FAA** (fetch-and-add) count as both an acquire-read and a release-write at the same time.

2.2 Explicit Epoch Persistency

We use the examples in Figs. 2b to 2d to explain the memory model we use. The notation $e \circlearrowleft e_r$ denotes execution of e with e_r configured as recovery code.[3] We use the *explicit epoch persistency* model by Izraelevitz et al. [2016]. As they argue this persistency model is a slight generalization of the x86 and ARM machine level persistency models. The model includes three operations to manage the persist order: an explicit flush, **flush** (also called a write-back), an asynchronous fence, **fence**, and a synchronous fence, $\mathbf{fence}_{\mathbf{sync}}$. In the absence of these instructions, *no guarantees* are given on the persist order. For instance, it is not safe to run the left-hand side of Fig. 2a with the recovery code in Fig. 2b. As there are no flushes or fences, the two writes might persist in any order: after a crash the recovery code might see y being 1 and x still being 0, even though, during normal execution, this would never be observable due to the release-write.

[2]The largest deviation from C11 is that we make no attempt to rule out data races on non-atomics which is undefined behavior in C11. This can be done with a *race-detector* [Dang et al. 2020; Kaiser et al. 2017]—we avoid that here for simplicity.
[3]Note, that this is *not* syntax in the programming language.

$$\textsc{Htc-atomic}\quad \frac{\text{atomic}(e) \qquad P \twoheadrightarrow Q_c}{\{P\}\, e\, \{Q \wedge Q_c\} \vdash \{P\}\, e\, \{Q\}\{Q_c\}} \qquad\qquad \textsc{Htr-idempotence}\quad \frac{\{P\}\, e\, \{Q\}\{Q_r\} \qquad Q_r \twoheadrightarrow \langle \text{PC} \rangle\, R \qquad \{R\}\, e_r\, \{Q_r\}\{Q_r\}}{\{P\}\, e \circlearrowleft e_r\, \{Q\}\{Q_r\}}$$

Fig. 3. Key rules for quadruples in Perennial

To enforce a certain persist order one must *explicitly* flush writes and then end an *epoch* with a fence. An asynchronous fence ensures that all writes that have been flushed before the fence persist prior to any writes after the fence. The asynchronous fence does *not* ensure that the flushed writes have actually been persisted; hence, if a crash happens after the fence, the writes flushed prior to it might still be lost. But, when a certain persist order has been established, recovery code can perform a kind of "backwards reasoning". For instance, in Fig. 2c the flush and fence implies that the write to x persists before the write to y. Hence the recovery code can read y, and then, if the read yielded 1, reason backwards through the persist order and conclude that it is now certain to read 37 from x. This makes the assertion in Fig. 2c safe. A synchronous fence, is stronger, but also potentially slower, than an asynchronous fence. It additionally *blocks execution* until all flushed writes have actually reached persistent memory. This means that had Fig. 2c used a synchronous fence, then the write to x would have been persisted with certainty after executing the program.

Flushes and fences interact with release-writes and acquires-reads as a way to "connect" the weak memory order and the persist order. If an acquire-read synchronizes with a release-write then anything flushed *and* fenced prior to the release-write is guaranteed to persist before anything following a fence after the acquire-read. In the durable MP example in Fig. 2b this ensures that the write to z in the right thread must persist after the write to x in the left thread and hence that the assertion made at recovery is safe. Note that the fence after the acquire-read of y is necessary. When performing an acquire-read a thread immediately gains knowledge of the writes the releasing thread know about. But, only after a fence does it gain knowledge about flushed and fenced writes known to the releasing thread. Note also that flushes without fences provide no ordering guarantees with respect to atomic operations.

The optimized durable MP example in Fig. 2d is similar to the durable MP example except that the left thread does not flush the write to x before sending it through y. Hence, when the right threads read 1 from y it is still certain to know about the write to x (as in Fig. 2a), but it no longer receives knowledge about the write being flushed. Hence, the right thread must flush x. With this being done it is still the case that the write to z persists after the write to x. But, it is no longer the case that the write to x will persist before the write to y. This brings us to the crucial point regarding this example: reading 1 from y carries with it different information to a concurrent thread (which gains knowledge that x holds 37) than it does to recovery code (which gains nothing). In Fig. 2b it would also have been safe for the recovery code to read y instead of z, but here this would not be safe. At the logic level, this means that the resources associated with the write to y in Fig. 2d must change at a crash, but it need not change in Fig. 2b.

The next section introduces Spirea. Readers who first want to see the full formal operational semantics can read Appendix A before proceeding.

3 BACKGROUND: CRASH REASONING FEATURES IN PERENNIAL

Perennial extends Hoare logic with a *crash Hoare quadruple* of the form $\{P\}\, e\, \{Q\}\{Q_c\}$. Here P and Q are standard pre- and postconditions. The fourth component Q_c is a *crash condition* that must hold during every step of execution of e. Since Q_c holds at every step, if a crash occurs at some point, then Q_c will necessarily hold at that point. Hence, the crash-condition is a property that recovery code can rely on after a crash.

In addition to standard language independent structural rules (a frame rule, a bind rule, *etc.*), the key rule for deriving a crash Hoare quadruple is Htc-atomic seen in Fig. 3. The rule states that to prove a crash Hoare quadruple for an *atomic* expression e, it suffices to prove that the pre-condition implies Q_c and an ordinary Hoare triple for e with Q_c added to the postcondition. Since e is atomic and can take only a single step, it suffices to show the crash condition before and after this single step. Note the use of the standard (non-separating) conjunction $\wedge$. This makes it possible to use all the resources one has at hand to show both Q and Q_c. This is a crucial aspect of crash conditions: they can be established without losing the resources necessary to show them.[4] The use of $\wedge$ is sound since, when the program runs, it will *either* take a normal step of execution (in which case the proof of Q is needed) *or* crash (in which case the proof of Q_c is needed). Since both cannot happen at the same time, it is not necessary to show the two conjuncts for disjoint resources. The Htc-atomic rule is important since it, in combination with the structural rules, allows us to show a crash Hoare quadruple by showing a normal Hoare triple at each step. This explains why we show rules for normal Hoare triples later in §4.

To show crash-safety Perennial offers *recovery Hoare quadruples* of the form $\{P\}\ e \circlearrowleft e_r\ \{Q\}\{Q_r\}$. The intuitive reading is: given that P holds initially, it is safe to execute e with the recovery program e_r. If e terminates in a value v without crashing then $Q(v)$ holds. If, on the other hand, one or more crashes occur during execution (of e and e_r) and e_r terminates in a value v, then $Q_r(v)$ holds.

Per the idempotence rule Htr-idempotence one can show a recovery Hoare quadruple for a program e and recovery program e_r by showing a crash Hoare quadruple for e and one for e_r. In both cases the crash condition is Q_r, such that e_r can rely on this resource; not directly though, as the crash itself might change Q_r, hence the inclusion of the post-crash modality. Since e_r itself maintains the crash condition Q_r, any number of crashes during e_r are still safe.

In summary, the proof burden for proving crash-safety is to pick a crash condition and apply Htr-idempotence. Then one verifies two crash Hoare quadruples. The verification of these is similar to using normal Hoare triples except that the crash condition must be shown at every step.[5]

4 SPIREA

Spirea is a CSL based on the Iris separation logic framework. As such it contains all the standard connectives from Iris-based separation logics such as the separating conjunction, ghost state, higher-order quantifiers, *etc.* For reasoning about programs it offers Hoare triples, recovery Hoare quadruples, and crash Hoare quadruples. The latter two support the same rules as they do in Perennial. In this section we explain the novel aspects of Spirea. Throughout the section we cover the verification of the two examples from Fig. 2a and Fig. 2c; proof outlines are shown in Fig. 7 and Fig. 8.

Knowledge vs. resources. In Iris a persistent proposition is one that does not entail ownership but only represents duplicable knowledge. $\square P$ means that P always holds, and a proposition P is persistent if $P \vdash \square P$. To avoid confusion with the different notions of "persistent" we use the word "knowledge" to mean persistent propositions. For example, $n = 37$ is knowledge and $\ell \hookrightarrow 37$ is not.

Conventions for modalities. As we will see, Spirea contains a healthy number of modalities. In order to avoid having to introduce a plethora of symbols, we denote modalities (except already well-known ones) as $\langle M \rangle$ where M is a mnemonic for the modality. All of our modalities satisfy

[4]This is in contrast to normal Iris invariants, where one has to sacrifice ownership of the resources necessary to show the invariant.

[5]Showing the crash condition is usually trivial and can be automated with a Coq tactic.

MOD-SEP
$$\langle \mathsf{M} \rangle\, P * \langle \mathsf{M} \rangle\, Q \vdash \langle \mathsf{M} \rangle (P * Q)$$

MOD-MONO
$$\frac{P \vdash Q}{\langle \mathsf{M} \rangle\, P \vdash \langle \mathsf{M} \rangle\, Q}$$

MOD-INTRO
$$P \vdash \langle \mathsf{M} \rangle\, P$$

MOD-IDEMP
$$\langle \mathsf{M} \rangle \langle \mathsf{M} \rangle\, P \dashv\vdash \langle \mathsf{M} \rangle\, P$$

MOD-ELIM
$$\langle \mathsf{M} \rangle\, P \vdash P$$

Fig. 4. General rules for modalities

LB-KNOWLEDGE
$$\frac{l \in \{\mathsf{p}, \mathsf{f}, \mathsf{s}\}}{\ell \gtrsim_{l} \sigma \vdash \Box\, \ell \gtrsim_{l} \sigma}$$

LB-PERSISTENT-FLUSH-STORE
$$\ell \gtrsim_{\mathsf{p}} \sigma \vdash \ell \gtrsim_{\mathsf{f}} \sigma \vdash \ell \gtrsim_{\mathsf{s}} \sigma$$

OBJ-NOFLUSH-NOBUFFER
$$\langle \mathsf{obj} \rangle\, P \vdash \langle \mathsf{NF} \rangle\, P \vdash \langle \mathsf{NB} \rangle\, P$$

MAPSTO-STORE-LB
$$\ell \hookrightarrow_{a} \vec{\sigma}\sigma \vdash \ell \gtrsim_{\mathsf{s}} \sigma$$

MAPSTO-LB-PERS
$$\frac{\sigma_2 \not\sqsubseteq \sigma_1 \qquad \ell \gtrsim_{\mathsf{p}} \sigma_2 \qquad \ell \hookrightarrow_{\mathsf{na}} \sigma_1 \vec{\sigma}}{\ell \hookrightarrow_{\mathsf{na}} \vec{\sigma}}$$

MAPSTO-NA-STORE-LB
$$\frac{\ell \gtrsim_{\mathsf{s}} \sigma_1 \qquad \ell \hookrightarrow_{\mathsf{na}} \vec{\sigma}\sigma_2}{\sigma_1 \sqsubseteq \sigma_2}$$

POST-FENCE-NO-FLUSH
$$\langle \mathsf{PF} \rangle \langle \mathsf{NF} \rangle\, P \vdash P$$

PFS-PF
$$\langle \mathsf{PF} \rangle\, P \vdash \langle \mathsf{PF_S} \rangle\, P$$

REC-IN-IF-REC
$$\mathrm{crashedIn}(\ell, \sigma) * \langle \mathsf{ifRec} \rangle_{\ell}\, P \vdash P$$

Rules for the post-crash modality

PC-NA-MAPSTO
$$\ell \hookrightarrow_{\mathsf{na}} \sigma_1 \sigma_2 \cdots \sigma_n \vdash \langle \mathsf{PC} \rangle \langle \mathsf{ifRec} \rangle_{\ell}\, \exists i \leq n.\, \ell \hookrightarrow_{\mathsf{na}} \psi(\sigma_1)\psi(\sigma_2)\cdots\psi(\sigma_i) * \mathrm{crashedIn}(\ell, \sigma_i)$$

PC-AT-MAPSTO
$$\ell \hookrightarrow_{\mathsf{at}} \sigma \vdash \langle \mathsf{PC} \rangle \langle \mathsf{ifRec} \rangle_{\ell}\, \exists \sigma_r.\, \ell \hookrightarrow_{\mathsf{at}} \psi(\sigma_r) * \mathrm{crashedIn}(\ell, \sigma_r)$$

PC-INVARIANT
$$\boxed{\ell \mid \pi} \vdash \langle \mathsf{PC} \rangle \langle \mathsf{ifRec} \rangle_{\ell}\, \boxed{\ell \mid \pi}$$

PC-PCF
$$\langle \mathsf{PC} \rangle\, P \vdash \langle \mathsf{PCF} \rangle\, P$$

PC-PERSIST-LB
$$\ell \gtrsim_{\mathsf{p}} \sigma \vdash \langle \mathsf{PC} \rangle\, \ell \gtrsim_{\mathsf{p}} \psi(\sigma) * \exists \sigma_r \sqsupseteq \sigma.\, \mathrm{crashedIn}(\ell, \sigma_r)$$

PCF-FLUSH-LB
$$\ell \gtrsim_{\mathsf{f}} \sigma \vdash \langle \mathsf{PCF} \rangle\, \ell \gtrsim_{\mathsf{p}} \psi(\sigma) * \exists \sigma_r \sqsupseteq \sigma.\, \mathrm{crashedIn}(\ell, \sigma_r)$$

REC-IN-AGREE
$$\mathrm{crashedIn}(\ell, \sigma) * \mathrm{crashedIn}(\ell, \sigma') \vdash \sigma = \sigma'$$

Fig. 5. Selected rules for assertions and modalities in the logic

basic structural rules such as MOD-SEP and MOD-MONO seen in Fig. 4. Additionally, some modalities are monadic (they satisfy MOD-INTRO, *etc.*) or comonadic (they satisfy MOD-ELIM, *etc.*).

Crash-Aware Invariants. As mentioned, one of the key innovations in Spirea is *crash-aware invariants* (or just invariants for short when it is clear from the context that we are not talking about Iris invariants). We start things off with the definition. The definition uses concepts in Spirea that we have yet to see, but these can be ignored for now. We will refer back to, and provide explanations of, the definition throughout the section.

Definition 4.1. A *crash-aware invariant* π consists of: a set of states Σ, a preorder $\sqsubseteq$ on Σ, a *write assertion* $\phi : \Sigma \times \mathrm{VAL} \longrightarrow \mathrm{dProp}$ (dProp is the type of propositions in Spirea), and a *state-change* function $\psi : \Sigma \longrightarrow \Sigma$ that is monotone w.r.t. $\sqsubseteq$. The data must satisfy the following two conditions:

(1) $\forall \sigma \in \Sigma, v \in \mathrm{VAL}.\, \phi(\sigma, v) \vdash \langle \mathsf{NB} \rangle\, \phi(\sigma, v)$
(2) $\forall \sigma \in \Sigma, v \in \mathrm{VAL}.\, \phi(\sigma, v) \vdash \langle \mathsf{PCF} \rangle\, \phi(\psi(\sigma), v)$.

For an invariant π we refer to its components, say ϕ, with $\pi.\phi$, but more often we just write ϕ when it is clear from context which invariant the component is from.

In the logic every location ℓ is associated with a specific invariant π throughout its lifetime. This invariant is chosen dynamically when the location is allocated by using the rules HT-NA-ALLOC and HT-AT-ALLOC that appear in Fig. 6. In these rules the *invariant assertion* $\boxed{\ell \mid \pi}$ appears in the postcondition. It denotes the knowledge that ℓ is associated with π. On the first line of the proof outlines (Fig. 7 and Fig. 8) we see invariant assertions for both x and y. For such preexisting locations invariants can be picked at the beginning of the proof (we will see the details in §5.2).

Ht-flush
$$\{\ell \gtrsim_{\mathrm{s}} \sigma\}\ \mathbf{flush}\ \ell\ \left\{\langle \mathsf{PF}\rangle(\ell \gtrsim_{\mathrm{f}} \sigma) * \langle \mathsf{PF}_{\mathsf{S}}\rangle(\ell \gtrsim_{\mathrm{p}} \sigma)\right\}$$

Ht-fence-sync
$$\{\langle \mathsf{PF}_{\mathsf{S}}\rangle\, P\}\ \mathbf{fence}_{\mathbf{sync}}\ \{P\}$$

Ht-fence
$$\{\langle \mathsf{PF}\rangle\, P\}\ \mathbf{fence}\ \{P\}$$

Ht-na-alloc
$$\{\phi(\sigma, v)\}\ \mathbf{ref}_{\mathsf{na}}\ v\ \left\{\ell.\ \boxed{\ell \mid \pi} * \ell \hookrightarrow_{\mathsf{na}} \sigma\right\}$$

Ht-at-alloc
$$\{\phi(\sigma, v)\}\ \mathbf{ref}_{\mathsf{at}}\ v\ \left\{\ell.\ \boxed{\ell \mid \pi} * \ell \hookrightarrow_{\mathsf{at}} \sigma\right\}$$

Ht-na-read
$$\left\{\begin{matrix} \boxed{\ell \mid \pi} * \ell \hookrightarrow_{\mathsf{na}} \vec{\sigma}\sigma\, * \\ \left(\begin{matrix}\langle \mathsf{obj}\rangle\, \forall v.\, \phi(\sigma, v) \mathrel{-\!\!*} \\ Q(v) * \phi(\sigma, v)\end{matrix}\right)\end{matrix}\right\}\ !_{\mathsf{na}}\, \ell\ \left\{w.\ \ell \hookrightarrow_{\mathsf{na}} \vec{\sigma}\sigma * Q(w)\right\}$$

Ht-na-write
$$\left\{\begin{matrix} \boxed{\ell \mid \pi} * \ell \hookrightarrow_{\mathsf{na}} \vec{\sigma}\sigma\, * \\ \phi(\sigma_t, v_t) * \sigma \sqsubseteq \sigma_t \end{matrix}\right\}\ \ell :=_{\mathsf{na}} v_t\ \left\{\ell \hookrightarrow_{\mathsf{na}} \vec{\sigma}\sigma\sigma_t\right\}$$

Ht-at-read
$$\left\{\begin{matrix} \boxed{\ell \mid \pi} * \ell \hookrightarrow_{\mathsf{at}} \sigma\, * \\ \langle \mathsf{obj}\rangle\, \forall \sigma_r \sqsupseteq \sigma, v_r.\, \phi(\sigma_r, v_r) \mathrel{-\!\!*} Q(\sigma_r, v_r) * \phi(\sigma_r, v_r) \end{matrix}\right\}\ !_{\mathsf{at}}\, \ell\ \left\{v.\ \exists \sigma_r \sqsupseteq \sigma.\ \ell \hookrightarrow_{\mathsf{at}} \sigma_r * \langle \mathsf{PF}\rangle\, Q(\sigma_r, v)\right\}$$

Ht-at-write
$$\left\{\begin{matrix} \boxed{\ell \mid \pi} * \ell \hookrightarrow_{\mathsf{at}} \sigma * \phi(\sigma_t, v_t) * \sigma \sqsubseteq \sigma_t\, * \\ \forall \sigma_c \sqsupseteq \sigma, v, v_c.\, \phi(\sigma, v) \mathrel{-\!\!*} \phi(\sigma_t, v_t) \mathrel{-\!\!*} \phi(\sigma_c, v_c) \mathrel{-\!\!*} \sigma_c \sqsubseteq \sigma_t \sqsubseteq \sigma_c \end{matrix}\right\}\ \ell :=_{\mathsf{at}} v_t\ \left\{\ell \hookrightarrow_{\mathsf{at}} \sigma_t\right\}$$

Fig. 6. Selected program rules for memory operations

```
$\left\{\begin{matrix} \boxed{x \mid \pi_x} * \boxed{y \mid \pi_{y,mp}}\, * \\ x \hookrightarrow_{\mathsf{na}} [\bot] * y \hookrightarrow_{\mathsf{at}} \bot * \mathrm{tok}_1 \end{matrix}\right\}$
$\left\{\begin{matrix} x \hookrightarrow_{\mathsf{na}} [\bot]\, * \\ y \hookrightarrow_{\mathsf{at}} \bot \end{matrix}\right\}$          ||  $\{y \hookrightarrow_{\mathsf{at}} \bot * \mathrm{tok}_1\}$
$x :=_{\mathsf{na}} 37;$                                        ||  if $!_{\mathsf{at}}\, y = 1$
$\{x \hookrightarrow_{\mathsf{na}} [\bot, \top]\}$                      ||  then
$y :=_{\mathsf{at}} 1$                                          ||    $\left\{\begin{matrix} y \hookrightarrow_{\mathsf{at}} \top\, * \\ x \hookrightarrow_{\mathsf{na}} [\bot, \top] \end{matrix}\right\}$
$\{y \hookrightarrow_{\mathsf{at}} \top\}$                               ||    assert $!_{\mathsf{na}}\, x = 37$
                                                 ||    $\{\mathrm{True}\}$
$\{\mathrm{True}\}$
```

Fig. 7. Proof outline for the message passing example.

```
$\left\{\begin{matrix} \boxed{x \mid \pi_x} * \boxed{y \mid \pi_{y,\mathit{ff}}}\, * \\ x \hookrightarrow_{\mathsf{na}} [\bot] * y \hookrightarrow_{\mathsf{na}} [\bot] \\ y \gtrsim_{\mathrm{p}} \sigma_y * y \hookrightarrow_{\mathsf{na}} [\sigma_y] \end{matrix}\right\}$
$x :=_{\mathsf{na}} 37;$
$\{x \hookrightarrow_{\mathsf{na}} [\bot, \top] * x \gtrsim_{\mathrm{s}} \top\}$
flush $x$;
$\{\langle \mathsf{PF}\rangle\, x \gtrsim_{\mathrm{f}} \top\}$
fence;
$\{x \gtrsim_{\mathrm{f}} \top * y \hookrightarrow [\bot]\}$
$y :=_{\mathsf{na}} 1$
$\{y \hookrightarrow_{\mathsf{na}} [\bot, \top]\}$

    $\circlearrowleft$

$\left\{\begin{matrix} \exists \sigma_x, \sigma_y. \\ \boxed{x \mid \pi_x} * \boxed{y \mid \pi_{y,\mathit{ff}}}\, * \\ x \gtrsim_{\mathrm{p}} \sigma_x * x \hookrightarrow_{\mathsf{na}} [\sigma_x]\, * \\ y \gtrsim_{\mathrm{p}} \sigma_y * y \hookrightarrow_{\mathsf{na}} [\sigma_y] \end{matrix}\right\}$
if $!_{\mathsf{at}}\, y = 1$
then
  $\{\sigma_y = \top * x \gtrsim_{\mathrm{f}} \top\}$
  $\{x \hookrightarrow_{\mathsf{na}} [\top]\}$
  assert $!_{\mathsf{na}}\, x = 37$
  $\{\mathrm{True}\}$
```

Fig. 8. Proof outline for the asynchronous fence example

The invariant assertions hold throughout the proofs, but to avoid clutter in the outlines we do not repeat unchanged resources.

Invariant States. Consider a thread reading y in parallel with the sending thread in Fig. 7. Such a thread can observe the initial value of 0, the final value of 1, and once it sees the latter it never sees the former again. We can represent the situation with a state transition system (STS): y can be in one of the two states $\bot$ and $\top$ (corresponding to 0 and 1 respectively) and it can transition from $\bot$ to $\top$—we say that $\top$ is a greater state and write $\bot \sqsubseteq \top$. A key insight going back to GPS is that the above can be put to good use in a logic by letting each location be governed by an STS as part of its invariant. This is the purpose of Σ and $\sqsubseteq$ in Def. 4.1, they represent an STS that the location must evolve through. In the examples, we use the described STS with two states for x and y as both locations are written exactly once. When writing to a location a state $\sigma \in \Sigma$ must be picked such that the states *grow monotonically* with each write. For a single location the memory model ensures all threads observe writes to it in the same order, and the invariant rules ensure that this order corresponds to an increasing order of states. Furthermore, while the weak memory order and the persist order do not agree in general they do coincide for a single location. We hence observe that we can soundly adopt the use of STSs for persistent memory such that they represent both the weak memory order (as in GPS) as well as the persist order.

Write Assertions. A release-write can transfer resources from one thread to another, as in Fig. 7 where the write to y carries with it the right to access x. The *write assertion* in invariants describe such resources. A write assertion, $\phi : \Sigma \times \textsc{Val} \rightarrow \mathrm{dProp}$, is parameterized over the invariant's states and values. The idea is that for *every* write to the location governed by the invariant, say with value v and state σ, the assertion $\phi(\sigma, v)$ holds.

As a simple example, in both Fig. 7 and Fig. 8 we pick the following write assertion for x:

$$\phi_x(\sigma, v) \triangleq (\sigma = \bot * v = 0) \vee (\sigma = \top * v = 37). \tag{1}$$

The write assertion gives *meaning* to the states by establishing a correspondence between them and specific values. Having the state determine the value in this way is a common pattern. Since x is not used for resource transfer this suffices for its write assertion. When we verify the message passing example below we see an example where resource transfer is needed.

Points-To Predicates. Points-to predicates in Spirea have the form $\ell \hookrightarrow_a \vec{\sigma}$. Here $\vec{\sigma}$ is a *sequence* of states that has been written to ℓ.[6] When a is na (respectively at) we say that the points-to predicate is non-atomic (atomic) and the location can then only be accessed using the na (at) access mode. On the first line in Fig. 7 we use a non-atomic points-to predicate for x and an atomic one for y.

The non-atomic points-to predicate entails exclusive ownership over ℓ and supports fractional permissions, denoted $\ell \hookrightarrow^q_{\mathsf{na}} \vec{\sigma}$ for a fraction $q \in (0; 1]$. (As usual, we often omit the fraction q if it is 1.) Hence, in Fig. 7 we need to transfer ownership over the points-to predicate for x from the left thread to the right thread. The sequence $\vec{\sigma}$ contains (at least) all writes that can ever be read again, both before and after a crash. It may be surprising that we use a sequence of states for non-atomics as prior logics for weak memory have been able to establish "normal" points-to predicates for non-atomics that associate a location with a single value, thereby completely hiding the weak semantics. However, this is not possible in the persistent setting, where the asynchronicity of writes and the fact that crashes are ever-present (in contrast to data-races that can be avoided) means that at least some old states must be remembered. For instance, in Fig. 8, at the end of executing the right thread we have the resource $x \hookrightarrow_{\mathsf{na}} [\bot, \top]$. This preserves the precise information that after a crash x can have the value 0 or 37.

The atomic points-to predicate does not entail ownership and is knowledge. Hence, several threads can access atomic locations in parallel. This is needed for y in Fig. 7 where both threads own $y \hookrightarrow_{\mathsf{at}} \bot$ initially. Since several threads can write to an atomic location without any synchronization the sequence of states $\vec{\sigma}$ is only *partial*. Other threads may have performed writes that the current thread is not aware of and that are thus not in $\vec{\sigma}$. Hence, for the atomic points-to predicate states can freely be dropped and, in practice, it often suffices to remember only the latest write. Therefore, and as the rules that take advantage of the entire sequence of states in the atomic case are fairly involved, in the remainder of the paper we only use the atomic points-to predicate with a single state and present specialized proof rules for this simpler case.

Message Passing Example. The message passing example contains reads and writes of all kinds. This makes it a great example to explain the read and write rules in Spirea and to see how invariants

[6]By convention, we name sequences with arrows $\vec{\sigma}$ and use juxtaposition for concatenation. For instance, $\vec{\sigma}\sigma$ is a sequence starting with $\vec{\sigma}$ and ending with σ. For a concrete sequence we sometimes use list notation, as in $[\sigma_1, \sigma_2, \sigma_3]$.

facilitate resource transfer between threads. We start with the write assertion for y:

$$\phi_{y,mp}(\bot, v) \triangleq v = 0 * \mathrm{tok}_0 \qquad \phi_{y,mp}(\top, v) \triangleq v = 1 * (x \hookrightarrow_{\mathsf{na}} [\bot, \top] \vee \mathrm{tok}_1)$$

The equalities on v should be clear. The two tokens, tok_0 and tok_1, are *exclusive*: Only one of each exist and hence $\mathrm{tok}_n * \mathrm{tok}_n$ is a contradiction (*i.e.,* it implies false). This is a standard construction using Iris ghost state. The purpose of these tokens and the disjunction is best explained in the proof.

Notice how we split the initial resources from the first to the second line in Fig. 7. The left thread gets the non-atomic points-to predicate for x and the right thread gets the token tok_1. The rest is knowledge, so both threads get a copy. We now cover the two writes and the two reads.

Non-atomic write ($x :=_{na} 37$). The rule Ht-na-write states that to write v to a non-atomic location one must pick a target state σ_t. We choose $\top$. The precondition requires an invariant assertion, a points-to predicate, that the write assertion holds, and that the new state preserves the order of the states. All of these are trivial: we have an invariant assertion, a points-to predicate ending in the state $\bot$, $\phi_x(\top, 37)$ is immediate from the definition in eq. (1), and $\bot \sqsubseteq \top$ per definition. In the postcondition we receive an updated points-to predicate with the newly written state appended at the end. Non-atomic writes are usually this trivial, as precise information about them is known.

Atomic write ($y :=_{at} 1$). The first line of the precondition of Ht-at-write is similar to what we just saw for non-atomics. We pick the state $\top$ for the write and show the write assertion by choosing the left side of the disjunction and using our points-predicate for x. That is, we transfer ownership over x into the invariant. The conjunct on the second line of the precondition of Ht-at-write serves to maintain the monotone order of writes. Since atomic locations can be shared, we need to account for potential racy writes to the location. The universally quantified σ_c represents such a write and the obligation is to show that it and the written state σ_t can transition between each other, $\sigma_c \sqsubseteq \sigma_t$ and $\sigma_t \sqsubseteq \sigma_c$. This ensures that they are equivalent w.r.t. the preorder and that the order of the states is preserved no matter which of the two racy writes end up first in the memory order. To show this obligation the writer can assume the assertion of both the original state σ, the concurrent state σ_c, and the written state σ_t. If we look at the whole program we are verifying it is clear that there are no concurrent writes to y. But, as we are verifying the left thread modularly in isolation, we must be able to draw this conclusion based solely on the invariant. To this end, we assume some concurrent write σ_c and must show $\sigma_c \sqsubseteq \top \sqsubseteq \sigma_c$. If $\sigma_c = \top$ the conclusion is trivial. If the $\sigma_c = \bot$ the conclusion is impossible. Fortunately, in this case we have the invariant for $\bot$ *twice*, hence we have the token tok_0 twice, which is a contradiction. Intuitively, the token tok_0 represents the right to write $\bot$ to x, and since only one token exists, this state can only ever be written once.

Atomic read ($!_{at}\, y$). Now in the right thread we, apply Ht-at-read. At the present time we can ignore the $\langle\mathrm{obj}\rangle$ and $\langle\mathrm{PF}\rangle$ in the rule. We have the invariant and the points-to predicate required in the precondition. The last conjunct lets us open the invariant, access its content, and potentially transfer resources in and out of the invariant. The resource Q represents the resources that we want to *transfer out* of the invariant. We use

$$Q(\bot, v) \triangleq v = 0 \qquad Q(\top, v) \triangleq v = 1 * x \hookrightarrow_{\mathsf{na}} [\bot, \top].$$

Hence, if we read 1 we transfer the points-to predicate for x out. We need to show the wand in Ht-at-read. For some read state σ_r and value v_r the reader receives the invariant $\phi(\sigma_r, v_r)$ (the antecedent of the wand). We now have access to the content of the invariant, but, since the invariant also appears in the consequent the access is temporary—we say that we have to *close* the invariant. If $\sigma_r = \bot$ then $Q(\bot, v_r)$ is plain knowledge and showing it and the invariant is trivial. If $\sigma_r = \top$ then we use $x \hookrightarrow_{\mathsf{na}} [\bot, \top]$ to show $Q(\top, v_r)$. However, now we can not use this points-to predicate to close the invariant. Fortunately, the invariant contains a disjunction and we can show the right disjunct using the tok_1 that the right thread owns. That is, we transfer tok_1 *in to* the invariant in order to transfer $x \hookrightarrow_{\mathsf{na}} [\bot, \top]$ *out of* the invariant. This sort of reasoning is well-known to readers familiar with Iris invariants, but it is in fact significantly stronger than the read rule in GPS and iGPS. In these logics, a read can only transfer *knowledge* out of the invariant—transferring ownership over resources is not possible! Returning to the proof, having shown the preconditions for the read, we now get Q in the postcondition. The case where we read 0 is trivial, so we consider the case where we read 1 and enter the branch. In this case we have the points-to predicate for x after the read, as shown in the proof outline. All that remains is to show that the read of x yields 37.

Non-atomic read ($!_{na}\, x$). We apply Ht-na-read which is much like the read rule for atomics, which we just went through. The notable difference is that for a non-atomic is it certain that the last state in the points-to predicate (σ in the rule) is read. Hence, the rule does not quantify over some read state. When applying the rule we pick $Q(v) \triangleq v = 37$, which is easy to show when opening the invariant, and which gives us what we need.

RMW Operations. We have now seen the rules for reading and writing. Spirea also contains rules for the RMW operations **CAS** and **FAA**. We do not include these rules for space reasons and since they are rather complex. Since RMW operations are simultaneously both a read and a write, our rules for these essentially combine the read and the write rule. The rules require that the write assertion is shown for the read value (like Ht-at-read) *and* the written value (like Ht-at-write). This is in contrast to other CSLs for weak memory, where the equivalent notion to our write assertion would not have to be shown for the read value. This makes resource transfer through RMW operations more restricted, but ensures that invariants are sound. In §6.2 we show how to combine Spirea with BaseSpirea for examples where the CAS rule is not strong enough, in §6.3 we see an example where the CAS rule is sufficient, and we discuss the limitation further in §7.

Flushes and Fences. To verify programs using flushes and fences we need assertions that capture the knowledge gained by these operations. Consider the pre-crash code in Fig. 8. Just after writing to x the thread merely knows that the write with state $\top$ exists (which implies that a successive read reads this or a more recent state). Knowledge of this form is captured by the *store lower bound* assertion $\ell \gtrsim_{\mathsf{s}} \sigma$. The program then flushes x and carries out an asynchronous fence. After this the thread knows that the write will persist before any succeeding writes. This form of knowledge is represented by the *flush lower bound* $\ell \gtrsim_{\mathsf{f}} \sigma$. Suppose the program had instead carried out a *synchronous* fence. The thread would then know that the write had been saved to persistent memory. The *persist lower bound* $\ell \gtrsim_{\mathsf{p}} \sigma$ represents this knowledge.

These assertions are lower bounds, in the sense that $\ell \gtrsim_{l} \sigma$ implies knowledge of a write in at least state σ but not that this is necessarily the most recent state. This, together with the fact that states grow monotonically, makes the assertions knowledge (lb-knowledge). The three lower bound relations are ordered as shown in lb-persistent-flush-store since a state is written before it is flushed, and since a synchronous fence is strictly stronger than an asynchronous fence.

Following the above, the effect of flushing a location ℓ and a fence is then that the most recent write σ known to the flushing thread advances from $\ell \gtrsim_{\mathsf{s}} \sigma$ to $\ell \gtrsim_{\mathsf{f}} \sigma$ (in the case of an asynchronous fence) or to $\ell \gtrsim_{\mathsf{p}} \ell$ (in the case of a synchronous). The rules for flush and fence should achieve this while taking the following three things into account: (1) **flush** and **fence** are two separate operations and the fence may not necessarily immediately follow the flush. (2) A fence can apply to arbitrarily many preceding flushes. (3) A fence is not only used in combination with a flush. As in Fig. 2b it is also used in combination with an acquire-read to acquire persist information from the release-write. We want our program rules to support all these usage patterns. To this end Spirea includes two *fence modalities*: $\langle \mathsf{PF} \rangle$ and $\langle \mathsf{PF_S} \rangle$. The assertions $\langle \mathsf{PF} \rangle\, P$ and $\langle \mathsf{PF_S} \rangle\, P$ mean that P holds after the next asynchronous fence and synchronous fence, respectively.

In Fig. 8 we apply Ht-flush at the **flush** operation. The precondition takes a store lower bound that we can extract from $x \hookrightarrow_{\mathsf{na}} [\bot, \top]$ using mapsto-store-lb. The postcondition contains both a flush lower bound under $\langle \mathsf{PF} \rangle$ and a persist lower bound under $\langle \mathsf{PF_S} \rangle$ such that the **flush** can later be matched with both types of fences. In our case we only need the flush lower bound. At **fence** we use Ht-fence. This rule (and Ht-fence-sync) exactly matches the intuition of the fence modalities. If P holds under a fence modality, then executing a fence eliminates the modality. In our case this means that we have the flush lower bound after the fence. Note, that since the fence modalities are modalities and have a separation rule (as mod-sep) the result from several flushes can be combined and extracted with a single fence. In the rule Ht-at-read the extracted resource Q is under a fence modality which enforces that a fence be used when necessary. As such, using modalities for fences neatly achieves the requirements stated above.

To conclude the proof of the pre-crash program in Fig. 8 we define the write assertion for y

$$\phi_{y,\mathit{ff}}(\sigma, v) \triangleq (\sigma = \bot * v = 0) \lor (\sigma = \top * v = 1 * x \gtrsim_{\mathsf{f}} \top). \tag{2}$$

The assertion contains a flush lower bound for x when y has the state $\top$. To prove this at the write to y we use the flush lower bound gained from the flush and the fence. In the next section we see how this is used to verify the recovery code.

Non-Deterministic Post-Crash Modality. To verify the entire flush and fence example, including the recovery code, we apply Htr-idempotence where we must pick a crash condition Q_c. The R in the rule is the precondition for the recovery code in Fig. 8. As a crash condition we pick $\langle \mathsf{PC} \rangle\, R$. Using the post-crash modality directly in the crash condition like this is common in Spirea as it turns out to be the most convenient approach in practice. Proving the wand for R in Htr-idempotence becomes trivial, and the proof effort is concentrated on showing the crash condition at every step. In order to do this, we need to understand how our post-crash modality works. The rules for it appear in the lower half of Fig. 5.

Consider how an invariant $\boxed{\ell \mid \pi}$ should change at a crash. As we have mentioned, our invariants are crash-aware, and we want them to survive crashes. At the same time our programming language supports allocation, and since allocations might not persist before a crash, locations can be entirely lost at crashes. If a location is not lost after a crash, we say that it was *recovered* after the crash, and only in this case would it make sense still to have an invariant assertion for it. Such a situation is common, and we capture it by an *if-recovered* modality: the assertion $\langle \mathsf{ifRec} \rangle_\ell\, P$ mean that if the location ℓ was recovered at the last crash, then P (which would typically mention ℓ) holds. The rule PC-invariant is now clear: it preserves invariants for locations as long as they are recovered.

The if-recovered modality captures some of the non-determinism at a crash. Additional non-determinism is present in the rule PC-na-mapsto for non-atomic points-to predicates. Here the non-determinism is represented by the existential quantifier. The rule states that, for some i, only the first i states of the points-to predicate exist after the crash (ignore the ψ in the rule for now, it is explained later in the section). For state σ_i, the rule contains the assertion $\mathsf{crashedIn}(\ell, \sigma_i)$. The meaning of this assertion is that σ_i is the most recent recovered state for ℓ, which is exactly how it is used in the rule. Only one such state exists so two such assertion must agree on the state rec-in-agree. The $\mathsf{crashedIn}(\ell, \sigma_i)$ assertion also implies that ℓ was in fact recovered and it can thus be used to eliminate the if-recovered modality as seen in rec-in-if-rec.

The only way to know with certainty that a location will be recovered is through a persistent lower bound $\ell \gtrsim_{\mathsf{p}} \sigma$. Per PC-persist-lb a persistent lower bound is preserved across a crash (again, ignore ψ) and the most recent recovered state σ_r has to be at least σ. In contrast, a store lower bound clearly offers no knowledge after a crash as it only deals with the weak memory order. But what about a flush lower bound? A flush lower bound (and the **flush** and **fence** it represents) provides no knowledge of the state of the persistent memory, and as such it too has no meaningful interaction with the post-crash modality. Its effect is more subtle and only restricts the order of persists, as in the flush and fence example where the write to x persists before the write to y. To tease out this effect in the logic we introduce a *post-crash-flush* modality: $\langle \mathsf{PCF} \rangle\, P$ means that P holds after a crash *if* we are in the fortunate scenario where everything flushed and fenced actually reached persistent memory before the crash. In this case, a flush lower bound is just as good as a persist lower bound, and PCF-FLUSH-LB results in the same resources under the post-crash-flush modality as we saw in PC-PERSIST-LB. The post-crash-flush modality is weaker than the post-crash modality (PC-PCF) so the rules for the post-crash modality also applies to it.

The single place where we use the post-crash-flush modality is in the second condition for write assertions in the definition of invariants (Def. 4.1). This condition is necessary to make it possible to transfer invariants across a crash, *i.e.,* it is used to prove soundness of PC-INVARIANT. During this proof the write assertion ϕ must be established for the recovered state σ. Since σ was recovered, it must have persisted before the crash, and thus anything flushed and fenced prior to σ (that ϕ might know about) is also guaranteed to have persisted. As such, using the post-crash-flush modality in the condition is sufficiently strong, and allows us to use PCF-FLUSH-LB to show that ϕ holds for the recovered state. We note that, in our example, it is easy to show (using PCF-FLUSH-LB) that the second condition in Def. 4.1 does indeed hold for $\phi_{y,\mathit{ff}}$.

By using the rules for the post-crash modality it is now quite trivial to show the crash condition at every program point in the pre-crash code. And with the resources after the crash established, proving the recovery code is also straightforward. If reading 1 from y the recovery code learns that $\sigma_y = \top$ and acquires the resource $x \gtrsim_{\mathsf{f}} \top$ from the invariant. The flush lower bound can be weakened to $x \gtrsim_{\mathsf{s}} \top$ per LB-PERSISTENT-FLUSH-STORE, and combined with $x \hookrightarrow_{\mathsf{na}} [\sigma_x]$ the rule MAPSTO-NA-STORE-LB implies that $\top \sqsubseteq \sigma_x$, which in turn means that $\sigma_x = \top$. With that established reading x is sure to result in 37 just as what we saw in the message passing example.

Subjectivity. We now take a step back and consider an issue that we have so far swept under the rug. Propositions in Spirea can be *subjective*. That is, describe facts that are true from one thread's perspective, but that are not necessarily true from the point of view of other threads. For instance, after the left thread in Fig. 8 has flushed x it knows $\langle \mathsf{PF} \rangle\, x \gtrsim_{\mathsf{f}} \top$. But, as a flush by one thread provides no orderings across threads, it would be unsound to transfer this resource to another thread. We thus need to make certain restrictions on resource transfer. We accomplish this with three comonadic modalities. The *no-buffer modality*, $\langle \mathsf{NB} \rangle\, P$, means that P does not contain any of the post-fence modalities.[7] The first condition in Def. 4.1 uses this modality to ensures that the described unsound transfer is not possible. Write assertions that involve $\langle \mathsf{PF} \rangle$ or $\langle \mathsf{PF_S} \rangle$ do not pass this requirement. The *no-flush modality*, $\langle \mathsf{NF} \rangle\, P$, adds the requirement that P does not contain knowledge of flushes $\ell \gtrsim_{\mathsf{f}} \sigma$. Assertions of the form $\langle \mathsf{NF} \rangle\, P$ are of interest as they can safely be extracted from the post-fence modality per POST-FENCE-NO-FLUSH. This is what allowed us to ignore the $\langle \mathsf{PF} \rangle$ modality when we applied HT-AT-READ in Fig. 7 as the Q we picked did not use flush lower bounds. Finally, the *objectively modality*, $\langle \mathsf{obj} \rangle\, P$, means that P holds at *all* points of view of the memory and thus that it is always sound to transfer P between threads. Examples are $\ell \gtrsim_{\mathsf{p}} \sigma$ and $\boxed{\ell}\boxed{\pi}$. One use of this modality is in HT-AT-READ where it ensures that the reading thread can not transfer subjective resources to other reading threads.

State-Change Function. The final component of invariants that we still have not seen is *state-change functions*. To understand the need for these, consider how we would verify the optimized message passing example in Fig. 2d. Similar to the verification in Fig. 8, the write to z need

[7]The name refers to the fact that flushes use a buffer in the operational semantics.

to carry with it the knowledge $x \gtrsim_{\mathsf{f}} \top$. In order for the left thread to have this knowledge it must acquire $x \gtrsim_{\mathsf{s}} \top$ when reading y. As such, the write assertion for y must have the form $\phi_y(\top, v) \triangleq v = 1 * x \gtrsim_{\mathsf{s}} \top$. However, as there are no fences between the writes to x and y, if the *recovery code* were to read y it would be unsound for it to gain the knowledge $x \gtrsim_{\mathsf{s}} \top$. In other words, the write to y serves to transfer a resource to concurrently running threads that should not be available to recovery code. To capture this, a monotone *state-change function* ψ can change the state of a write after a crash. The idea is that if a write corresponds to the state σ before a crash, it then corresponds to $\psi(\sigma)$ after the crash. This is evident by looking at the crash related rules in Fig. 5 where states under the post-crash modality always have ψ applied to them. In examples where the above issue does not arise, the state-change function can simply be the identity function, and then the ψs can be ignored as we have done so far.

In order to verify the optimized message passing example we can extend the set of states for y with an additional state σ_{pc} that is below the two other states. The state-change function transitions every write into this state at a crash: $\psi(\sigma) \triangleq \sigma_{pc}$. The write assertion for this state is simply $\phi_y(\sigma_{pc}, v) \triangleq v = 0 \lor v = 1$. This ensures that if the recovery code were to read y it would gain no information whatsoever while still allowing for the desired resource transfer to work.

Summary. We have now completed our tour of Spirea. We hope it has become clear that it supports thread-local modular reasoning by extending ideas from separation logic, in particular ownership and resource transfer, with a range of modalities, which allow us to capture the subtle conditions under which resource transfer is sound.

5 SOUNDNESS

In this section we present an overview over the operational semantics of λ_{pmem}, state the soundness theorem of Spirea, and give an overview of the model, including some of the details. Readers who are more interested in seeing Spirea applied to examples can proceed to our case studies in §6.

5.1 Operational Semantics

The semantics of λ_{pmem} is a small-step interleaving operational semantics. Like prior such semantics for weak memory, it is based on *views*. For instance, Bila et al. created a view-based operational semantics for the x86 and ARM persistency models [Bila et al. 2022].

The small-step semantics is lifted to a big-step *recoverable execution relation* of the form $e_r; \rho \Rightarrow_{\mathsf{r}} \rho'; s$. Here, e_r is the recovery expression to execute after a crash, ρ and ρ' are machine configurations, and $s \in \{\mathsf{NotCrashed}, \mathsf{Crashed}\}$ is a crash-status. A machine configuration contains the state of entire machine, in particular the memory and all threads. The meaning of the relation is then: a machine in state ρ can execute to state ρ' with zero or more crashes along the way where e_r is executed after every crash. The crash-status indicates whether the execution has been crash free or not. If $s = \mathsf{NotCrashed}$ the execution was crash free and otherwise if $s = \mathsf{Crashed}$ then one or more crashed occurred. As we see below the soundness theorem is stated in terms of the recoverable execution relation.

The full operational semantics appears in Appendix A.

5.2 Soundness

Before we state the soundness theorem we define the safety result that the soundness theorem implies.

Definition 5.1 (safe). For expressions e and e_r, memory configuration M, and meta-level predicates on values Φ and Φ_r, $\mathsf{safe}(e, e_r, M, \Phi, \Phi_r)$ holds if, for any recoverable execution

$$e_r; \langle M, [\langle e, \langle \bot, \bot, \bot \rangle \rangle] \rangle \Rightarrow_{\mathsf{r}} \langle M, \vec{t} \rangle; s$$

it is the case that: (1) For every thread $\langle e, \mathcal{T} \rangle \in \vec{t}$, if e is not a value then the thread is not stuck. (2) For $\langle e', \mathcal{T} \rangle = (\vec{t})_1$, if e' is a value v (*i.e.,* the initial expression e terminated) then $\Phi(v)$ holds if $s = \mathsf{NotCrashed}$ and $\Phi_r(v)$ holds if $s = \mathsf{Crashed}$.

Theorem 5.2 (soundness). *Given expressions e and e_r, meta-level predicates on values Φ and Φ_r, a finite set of location L, and for each $\ell \in L$: an access mode a_ℓ, an invariant π_ℓ, a state $\sigma_\ell \in \pi_\ell.\phi$* (i.e., *an element of the state of the invariant π_ℓ). Let R be the resource*

$$\mathop{\Huge *}_{\ell \in \mathsf{dom}(h)} \boxed{\ell} \boxed{\pi_\ell} * \ell \gtrsim_p \sigma_\ell * \ell \hookrightarrow_{a_\ell} \sigma_\ell.$$

If $R \twoheadrightarrow \mathop{*}_{\ell \in \mathsf{dom}(h)} \pi_\ell.\phi(\sigma_\ell, v_\ell)$ and the recovery Hoare triple $\{R\}\, e \circlearrowleft e_r\, \{\Phi\}\{\Phi_r\}$ are provable in Spirea then $\mathsf{safe}(e, e_r, \langle h, \mathcal{P} \rangle, \Phi, \Phi_r)$ *holds where $h(\ell) = \langle v_\ell, \bot, \bot, \bot \rangle$ and where $\mathcal{P}(\ell) = 0$ for all $\ell \in L$.*

This theorem applies to a memory that is not necessarily empty to begin with. When applying the soundness theorem one then gets to pick, for each location, its access mode, invariant, initial state, *etc.* The resource R then contains the resources for all locations. It must then be shown that the invariants hold for the initial states, and to do this one can use R. This is such that the initial invariants can use resources (persistent lower bounds, points-to predicates, *etc.*) for other locations.

5.3 Model

We give a brief overview of the model of Spirea and highlight some of the underlying key ideas.

Overall Structure. Spirea is modeled atop a lower-level logic that we call BaseSpirea. BaseSpirea is constructed as an instantiation of Perennial's program logic framework based on the Iris base logic. This framework gives BaseSpirea basic definitions of the three Hoare triples/quadruples. Based on these we define various assertions to represent the physical state, define a post-crash modality, and prove program rules. However, these program proof rules directly expose the intricacies of the operational semantics, such as views, timestamps, and histories, and thus, while perfectly capable of verifying programs, BaseSpirea is quite tedious to use. We explain BaseSpirea in more detail in Appendix B. To provide the more abstract reasoning rules of Spirea, we use BaseSpirea to model Spirea. It is at this level that we add crash-aware invariants, the facilities for handling persistent memory instructions without explicit mention of views, and a post-crash modality that works for the higher-level assertions.

Crash-Aware Invariants. As mentioned in the introduction, a key challenge w.r.t. the model of Spirea's crash-aware invariants is that it is not clear how Iris invariants can be reconciled with crashes. We therefore take a different approach to invariants than other Iris-based logics for weak memory in that we do not model our crash-aware invariants using Iris invariants. Instead our model includes the resources for invariants inside the *state interpretation.* The state interpretation is a resource that is threaded through Hoare triples/quadruples in the program logic. With this approach the content of invariants is only available in the context of a Hoare triple/quadruple (as opposed to Iris invariants that can be accessed independently of a program). However, this is the case already in prior logics for weak memory, as accessing invariants in a weak memory model needs physical synchronization. The benefit of our approach is that when a crash occurs (more precisely, when proving soundness of Htr-idempotence), the resources belonging to all invariants are found inside the state interpretation, and can then be systematically updated to account for the crash.

Post-Crash Modality. We explain our post-crash modality with a simplified sketch of its model that highlights the key ideas.

$$[\![\langle \mathsf{PC} \rangle\, P]\!] \triangleq \lambda \mathcal{T}, \vec{\gamma}_{old}.\ \forall \vec{\gamma}_{new}.\ R(\vec{\gamma}_{old}, \vec{\gamma}_{new}) \twoheadrightarrow R(\vec{\gamma}_{old}, \vec{\gamma}_{new}) * [\![P]\!](\langle \bot, \bot, \bot \rangle, \vec{\gamma}_{new})$$

The semantic domain of propositions in Spirea is monotone predicates over thread views and a record of ghost names (denoted $\vec{\gamma}$). This explains why the model of the modality is a function taking two such arguments. Since resources are changed by a crash, new ghost resources along with new ghost names are introduced after a crash. The universal quantifier is over any such new record of new ghost names. However, the new resources are, to some extent, related to the old resources. The relationship is represented by the *exchange resource* R, which makes it possible to exchange old resources (valid before the crash) into new resources (valid after the crash). This works through rules of the form $P_{old} * R(\vec{\gamma}_{old}, \vec{\gamma}_{new}) \twoheadrightarrow P_{new} * R(\vec{\gamma}_{old}, \vec{\gamma}_{new})$. Here P_{old} could be a points-to predicate before the crash and P_{new} would then be an updated points-to predicate corresponding to the physical state after the crash. When proving soundness of a rule such as PC-NA-MAPSTO we then use the exchange resource to acquire the updated points-to predicate. Note that as R appears in the conclusion, it can perform these exchanges without being consumed itself. This is necessary to prove rules such as MOD-SEP for the post-crash modality. The definition of R is rather extensive as it must allow for resource exchanges for *all* the various resources used in the model. Establishing R is done in the soundness proof of HTR-IDEMPOTENCE. This rule is given an assumption involving a post-crash modality, and to extract the resource under it, R must be procured.

6 CASE STUDIES

In order to demonstrate the usefulness of our logic we have used it to verify several case studies.

6.1 Read-Optimized Reference

To show how Spirea supports modular specifications, we give in Fig. 9 a specification of a library implementing what we call *read-optimized references.* This module implements an interface that appears to clients as a single reference that can be read and writen. The implementation however optimizes the performance of reads. It does this by storing the content of the reference redundantly both in a "volatile" location (one can imagine it being stored in faster volatile memory) and in a persistent location (in the slightly slower persistent memory). When a client writes to the read-optimized reference the value it is saved to both locations, but when reading only the volatile reference is consulted for improved performance.

In the specification, an abstract (existentially quantified) predicate $\mathsf{isRR}(vr, v)$ is used to abstract over (hide from clients) the concrete data representation used by the library implementation; intuitively, it means that the value vr is a read optimized value with value v. After a crash, the volatile location might be lost and hence the reference needs to be recovered before it can be used after a crash. The abstract predicate $\mathsf{recRR}(vr, v)$ intuitively means that vr needs recovery. Just like in the verification of the flush and fence example we choose a crash condition that directly contains the post-crash modality. This simplifies the specification, in particular, in the crash condition for `write`. During the execution of `write`, after updating the volatile location but before updating the persistent location, the read-optimized reference is in an inconsistent state where it satisfies neither isRR for the old value nor the new value. Instead of trying to express this intermediate state we give the client what they actually need: the information that after a crash the read-optimized reference is recoverable in either the old or the new state.

Our Coq mechanization contains the full proof of the specification.

```
init ≜ λv.
  let per = ref_na v in
  flush per; fence_sync;
  let vol = ref_na v in
  (per, vol)
```

```
read ≜ λvr. !_na(π₂ vr)
write ≜ λvr, v.
  (π₁ vr) :=_na v;
  flush (π₁ vr); fence_sync;
  (π₂ vr) :=_na v
```

```
recover ≜ λvr.
  let per = π₁ vr in
  let vol = ref_na (!_na per) in
  (per, vol)
```

$$\{\mathrm{True}\}\ \mathtt{init}\ v\ \{vr.\ \mathrm{isRR}(vr, v)\}\{\mathrm{True}\} \qquad \{\mathrm{isRR}(vr, v)\}\ \mathtt{read}\ vr\ \{w.\ v = w * \mathrm{isRR}(vr, v)\}\{\langle \mathrm{PC}\rangle\ \mathrm{recRR}(vr, v)\}$$

$$\{\mathrm{isRR}(vr, v)\}\ \mathtt{write}\ vr\ w\ \{u.\ \mathrm{isRR}(vr, w)\}\{\langle \mathrm{PC}\rangle\ \exists u \in \{v, w\}.\ \mathrm{recRR}(vr, u)\}$$

$$\{\mathrm{recRR}(vr, v)\}\ \mathtt{recover}\ vr\ \left\{vr'.\ \mathrm{isRR}(vr', v)\right\}\{\langle \mathrm{PC}\rangle\ \mathrm{recRR}(vr, v)\} \qquad \mathrm{isRR}(vr, v) \vdash \langle \mathrm{PC}\rangle\ \mathrm{recRR}(vr, v)$$

Fig. 9. Implementation and specification of the read-optimized reference

```
makeStack ≜ λ_.
  let node = ref_na nil in
  flush node;
  fence;
  ref_at node
sync ≜ λtoHead.
  flush toHead;
  fence_sync;
nil ≜ inj₁ ()
cons v toNext ≜ inj₂ (v, toNext)
```

```
pop ≜ rec loop toHead =
  let head = !_at toHead in
  fence;
  match !_na head with
    inj₁ _ ⇒ inj₁ ()
    inj₂ pair ⇒
      let next = !_na(π₂ pair) in
      if CAS toHead head next
      then inj₂ (π₁ pair)
      else loop toHead
```

```
push ≜ λtoHead, val.
  let toNext = ref_na () in
  let newNode =
    ref_na (cons val toNext) in
  flush newNode;
  (rec loop () =
    let head = !_at toHead in
    toNext :=_na head;
    flush toNext; fence;
    if CAS toHead head newNode
    then () else loop ()) ()
```

Fig. 10. Implementation of the durable Treiber stack

6.2 Atomic Persists

Raad et al. [2020a] used the POG logic to verify an example where one thread writes to two locations, flushes and fences the writes, and transfers the information to a second thread through a spin lock. They call this example the *atomic persists* example. Due to the limitations of the **CAS** rule in Spirea we can not verify the spin lock in Spirea. Instead we verify the spin lock in BaseSpirea but give it a specification inside Spirea. We give the lock a crash-aware lock specification, similar to the one found in Perennial [Chajed 2022, Chapter 3]. With the lock verified in BaseSpirea we can then verify the rest of the example purely in Spirea. This demonstrates both how to use BaseSpirea in combination with Spirea and modularity. In the proof given by Raad et al. [2020a] the lock and the clients are verified together using one global invariant that contains knowledge about the locations used both internally in the lock and in the two clients. Hence, if the lock implementation is changed, the entire proof is affected. In our proof the lock is given a modular specification and a change in the lock implementation will only affect this proof and not the verification of the clients. For more details see Appendix D or the full proof in our Coq mechanization.

6.3 Durable Data-Structures With Null-Recovery

Concurrent *non-blocking* data structures have the property that they can be made durable and crash-safe by appropriately inserting flushes and fences [Friedman et al. 2020; Izraelevitz et al. 2016]. They furthermore enjoy *null-recovery*. As mentioned, this is the property that no recovery code is needed after a crash to restore the consistency of the data structure. Data structures with

$$\{\mathsf{True}\}\ \mathtt{makeStack}\ ()\ \{\ell.\ \mathsf{isStack}(\ell,\phi)\} \qquad \{\mathsf{isStack}(\ell,\phi) * \phi(w)\}\ \mathtt{push}\ \ell\ w\ \{\mathsf{True}\}$$

$$\{\mathsf{isStack}(\ell,\phi)\}\ \mathtt{pop}\ \ell\ \{v.\ v = \mathbf{inj}_1\ () \lor \exists x.\ v = \mathbf{inj}_2\ x * \phi(x)\} \qquad \{\mathsf{isStack}(\ell,\phi)\}\ \mathtt{sync}\ \ell\ \{\mathsf{synced}(\ell)\}$$

$$\mathsf{isStack}(\ell,\phi) \twoheadrightarrow \langle\mathsf{PCF}\rangle\,\mathsf{isStack}(\ell,\phi) \qquad \mathsf{isStack}(\ell,\phi) * \mathsf{synced}(\ell) \twoheadrightarrow \langle\mathsf{PC}\rangle\,\mathsf{isStack}(\ell,\phi)$$

Fig. 11. Specification of the durable Treiber stack

this property are *by construction* always in a consistent state—even after a crash. This makes them particularly well suited in a persistent setting and easier to use as clients of such data structures do not need to carry out recovery procedures (in contrast to, for instance, the read optimized reference). One would therefore hope to be able to derive similarly easy to use CSL specifications for such data structures. In this section we show how this is the possible in Spirea and explain how to specify and verify safety (including thread-safety and crash-safety) of non-blocking data structures with null-recovery. In our Coq mechanization we have verified durable implementations of both the Treiber stack and the Michael-Scott queue. These case studies show that our crash-aware invariants are sufficiently expressive to capture representation predicates for durable concurrent data structures and capable of handling null-recovery.

For space reasons we cover only the Treiber stack in this section. We focus on the resulting specification and sketch the proof. The full verification of both examples appears in our mechanization.

6.3.1 Implementation. The Treiber stack consists of a pointer to a linked list where, for thread-safety, the pointer is updated with **CAS**. The implementation of the stack appears in Fig. 10. We use pointers to sums to represent nodes in the linked list: $\mathbf{inj}_1\ ()$ represents a nil-node and $\mathbf{inj}_2\ (v,\ell)$ represents a cons-node with value v and with ℓ pointing to the succeeding node. In order to make the stack crash-safe we have inserted flushes and fences appropriately.

Our implementation is *buffered durable linearizable*, which means that it never waits (with $\mathbf{fence_{sync}}$) for an operation to reach persistent memory, but only ensures (with **fence**) that operations persist in the order in which they linearize. This improves performance but means that at a crash some returned operations might be lost. As is common for such data structures we include a `sync` operation that explicitly makes sure that the stack is persisted by using $\mathbf{fence_{sync}}$.

6.3.2 Specification. The specification (in Fig. 11) enforces that a predicate $\phi : \textsc{Val} \longrightarrow \mathsf{dProp}$ holds for each item in the stack. The specifications make use of an abstract (existentially quantified) representation predicate isStack, which is persistent, in the Iris sense, and hence duplicable, so that several threads can access the stack concurrently. Since isStack is persistent it does not need to appear in crash-conditions and hence we can use normal Hoare triples instead of crash Hoare triples. As such, the non-highlighted part forms a completely typical per-item CSL specification for a concurrent stack. This is exactly what we want, as it implies that a client can use the durable stack as they would a normal stack. Note that our specification does not imply linearizability or the LIFO property of the stack, but it does imply thread-safety and crash-safety.

The three highlighted rules are specific for persistent memory. The first of these shows that by running `sync` ℓ one gets the resource $\mathsf{synced}(\ell)$ which is evidence that the stack has been persisted. The two last rules concern the interaction between isStack and the post-crash modalities. The first rule states that if ℓ is a stack before a crash then after a crash it is still a stack, but only under the $\langle\mathsf{PCF}\rangle$ modality since the stack is buffered. The second rule applies if the stack is certain to have been persisted, as witnessed by synced; in this case the stack is preserved under the $\langle\mathsf{PC}\rangle$ modality.

The last two rules capture not only crash-safety but also the null-recovery property of the stack. They imply that with no recovery code needed, the isStack representation predicate can be reclaimed after a crash, and thus that a client can safely keep using the stack after a crash.

$$
\begin{aligned}
\mathsf{synced}(\ell) &\triangleq \ell \gtrsim_{\mathsf{p}} \star \\
\mathsf{isStack}(\ell, \phi) &\triangleq \boxed{\ell}\boxed{\pi_{stack}(\phi)} * \ell \hookrightarrow_{\mathsf{at}} \star \\
\phi_{stack}(\phi)(\_, v) &\triangleq \exists \ell_h, xs \in List(\textsc{Val}).\, v = \ell_h * \mathop{\Large\ast}_{x \in xs} \phi(x) * \mathsf{isNode}(\ell, xs) \\
\mathsf{isNode}(\ell_{node}, \quad []) &\triangleq \exists q.\, \boxed{\ell_{node}}\boxed{\mathtt{inj}_1\ ()} * \ell_{node} \hookrightarrow^{q}_{\mathsf{na}} \star * \ell_{node} \gtrsim_{\mathsf{f}} \star \\
\mathsf{isNode}(\ell_{node}, x :: xs) &\triangleq \exists \ell_{toNext}, \ell_{next}, q_1, q_2, \vec{\sigma}, i.\, \boxed{\ell_{node}}\boxed{\mathtt{inj}_2\ (x, \ell_{toNext})} * \ell_{node} \hookrightarrow^{q_1}_{\mathsf{na}} \star * \ell_{node} \gtrsim_{\mathsf{f}} \star * \\
&\qquad \boxed{\ell_{toNext}}\boxed{\pi_{toNext}} * \ell_{toNext} \hookrightarrow^{q_2}_{\mathsf{na}} \vec{\sigma}(i, \ell_{next}) * \ell_{toNext} \gtrsim_{\mathsf{f}} (i, \ell_{next}) * \mathsf{isNode}(\ell_{next}, xs)
\end{aligned}
$$

Fig. 12. Invariants and definitions used in the proof of durable concurrent stack

For the specification to be sound in our weak persistent memory setting, the per-item predicate ϕ must satisfy that for all $v \in \textsc{Val}$ it is the case that (1) $\phi(v) \vdash \langle \mathsf{NB} \rangle\, \phi(v)$, (2) $\phi(v) \vdash \langle \mathsf{PCF} \rangle\, \phi(v)$, and (3) $\phi(v) \vdash \Box\, \phi(v)$. The first two requirements are necessary to make ϕ safe to transfer between threads and across crashes. The third requirement expresses that ϕ must be persistent (in the Iris sense). This is required for a subtle reason: Since the stack is buffered, operations might return before they persist. Therefore, a value v can be popped from the stack (at which point the client is given $\phi(v)$), and then a crash can happen before the changes by the pop persist. Then, after the crash, v is still present in the stack, and thus it can be popped again (at which point the client is given $\phi(v)$ once more). In summary, due to crashes, the same value can be popped several times and hence the resource must be duplicable, *i.e.,* persistent. This requirement holds, for instance, for simple properties such as v being an even number and for assertions about atomic locations. Had the implementation been non-buffered, *i.e.,* implemented using the synchronous fence, then this requirement could be removed.

6.3.3 Proof (sketch). The proof proceeds by defining the predicates synced and isStack and then verifying that the specifications hold. The definition of synced expresses that ℓ has been persisted. For isStack we use three invariants. In all three the ψ function is the identity function. Two of the invariants use the abstract state set $1 = \{\star\}$. Elements of this abstract state carry no information, but lower bounds are still meaningful, *e.g.,* $\ell \gtrsim_{\mathsf{f}} \star$ means that location ℓ has certainly been flushed.

For a node the pointer to the sum never changes. For these locations we use the *constant invariant.* Given a value v the constant invariant $\pi_{const}(v)$ has the abstract state 1 and the invariant $\phi_{const}(\_, v') \triangleq v = v'$. We use the notation $\boxed{\ell}\boxed{v}$ for $\boxed{\ell}\boxed{\pi_{const}(v)}$.

The pointer from a cons-node to its successor potentially changes many times in push if the **CAS** in push fails. For this location we use the invariant π_{toNext}. Its abstract state is $\mathbb{N} \times \textsc{Val}$ ordered by the natural numbers in the first component. The invariant is $\phi_{toNext} = \lambda(n, v), v'.\, v = v'$.

For the stack itself (the pointer to the head of the linked list) we use the invariant $\pi_{stack}(\phi)$. Its abstract state is 1 and the invariant $\phi_{stack}(\phi)$ appears in Fig. 12. It states that there exists a logic-level list xs, all of whose elements satisfy ϕ, and it uses isNode to recursively express that the structure of the linked list corresponds to xs.

With these definitions and invariants in place the proof that the code satisfies the specification is fairly straightforward; see our Coq mechanization for the details.

We finally remark that a similar (non-persistent, non-weak memory) concurrent stack can be verified in standard Iris [Birkedal and Bizjak 2020]. The Iris proof uses an Iris invariant to define the isStack representation predicate.

7 RELATED AND FUTURE WORK

We now discuss aspects of related work that have not already been treated in the paper.

Logics for Persistent Memory. To our knowledge, there are only two prior program logics for persistent memory, namely Persistent Owicky-Gries (POG) [Raad et al. 2020a] and Pierogi [Bila et al. 2022]. Both POG and Pierogi focus on the persistent memory model of the x86 architecture [Raad et al. 2020a], which is stronger, both in terms of weak and persistent memory, than our memory model, which does not include details specific for any one architecture; instead it is a slight generalization of the persistent memory models in x86 and ARM. The programming languages covered by POG and Pierogi are much simpler than ours, λ_{pmem}; the languages in *op. cit.* support only a static number of threads running sequential commands, and a static number of memory locations. In contrast, λ_{pmem} includes more high-level features such as higher-order functions and dynamic allocation of threads and locations.

Both POG and Pierogi are Owicki-Gries-style program logics. POG makes use of rely-guarantee style reasoning to support composition of threads that do not interfere, whereas Pierogi does not support thread-local reasoning. In contrast, Spirea is a separation logic and hence it supports frame rules and thread-local reasoning. Moreover, since Spirea is built on top of Iris, it includes advanced features such as user-defineable ghost state and higher-order quantification, which are not present in POG or Pierogi but which are important for modular specification and verification of libraries, such as the stack case study we considered in §6.3. From Perennial we gain the ability to reason about durable resources in a convenient fashion using normal separation logic ownership.

In contrast to POG but similarly to Pierogi, our Spirea logic is mechanized in a proof assistant. Pierogi has been mechanized in Isabelle/HOL and its authors report that the Sledgehammer tool can be used to search automatically for program proof rules to apply. In contrast, we make use of the Iris Proof Mode [Krebbers et al. 2017] to support interactive development of program proofs in the Coq style, which works well for our higher-order logic and larger examples.

Similarly to Pierogi, Spirea supports reasoning directly about optimized flushes (write-backs) (**`flush`**) and the use of fences. In contrast, POG only supports reasoning about a stronger operation that combines the write back and the fence. To handle other programs they instead offer a translation that in some cases can translate a program with the weaker, and more tricky to reason about, instructions into equivalent programs. This translation only works for programs that use these instructions in a certain pattern, and programs that do not adhere to this pattern can not be reasoned about using their logic. Since we handle these operations directly we can verify such programs.

Finally, POG and Pierogi have, to the best of our knowledge, only been applied to reason about very small programs consisting of only a few lines, whereas we have used Spirea to verify larger programs, in particular entire data structures. Additionally we have shown how to give such data structures modular specifications as extensions of traditional CSL specifications.

Separation Logic for Weak Memory. GPS [Turon et al. 2014] is a program logic for the release-acquire and non-atomic fragment of the C11 weak memory model. The logic introduced *protocols* to reason about atomic location, the inspiration for our crash-aware invariants. GPS does not use protocols for non-atomic locations, but instead a standard points-to predicate. As mentioned, this approach is not sufficient in a persistent setting. The CAS rule in GPS does not require that (what we call) the invariant for the read value is preserved. When reading v_1 and simultaneously writing v_2 with a CAS, the CAS rules in GPS allows one to use the invariant for v_1 to show the invariant for v_2 and keep any additional resources *without* reestablishing the invariant for v_1. This is sound because the C11 semantics ensures that no CAS operation will ever read v_1 again. While this is also the case in our semantics, after a crash, the write for v_1 might have been persisted while the write for v_2 has not been persisted. Then another CAS operation might read v_1 again. Hence, in the presence of crashes the GPS CAS rule is unsound. Our rule for CAS requires that the invariant still holds for the read value, ensuring that the invariant always holds for all writes. This is sound even with crashes but is significantly more limiting than the GPS CAS rule. Essentially, the GPS CAS rule is sound for transferring resources between concurrently running CAS-operations, but not across crashes. Our CAS rule is sound for transferring resources across crashes, but only in a limited way between concurrent CAS'es. Creating a CSL rule that is simultaneously sound for both is very challenging and something that we would like to explore in future work. The CAS rule in BaseSpirea does not suffer from this limitation, and as demonstrated in §6.2, it can be used together with Spirea for cases where a stronger CAS rule would otherwise be needed.

The read rule for atomic locations in GPS does not make it possible to transfer exclusive resources out of the invariant for the value read. Our read rule makes it possible to extract exclusive resources as long as the invariant still holds (for instance by transferring other resources into the invariant). We make use of this capability to verify the message passing examples. In GPS an additional feature, *escrows*, is needed to verify the message passing examples.

Our use of modalities to reason about fences is inspired by Fenced Separation Logic (FSL), a program logic that supports reasoning about the release and acquire memory fences in the C11 memory model [Doko and Vafeiadis 2016]. FSL includes two fence modalities to describe resources that have been prepared for release or acquire by a release or acquire fence. The release and acquire fences in C11 serve a different purpose than those in λ_{pmem} and the modalities in FLS are correspondingly different as well.

Recently, in the context of weak memory we have seen logics that support specifications that go beyond safety. Compass [Dang et al. 2022] and Cosmos [Mével and Jourdan 2021] are both capable of showing stronger correctness results by using *logically atomic triples* as specifications. In contrast, our specification for the durable stack only implies safety. We think it would be interesting to investigate how ideas from these logics apply in our setting and we believe that a stronger CAS rule (per the discussion above) is necessary to achieve this.

Separation Logics for Durable Storage. Crash Hoare Logic [Chen et al. 2016] and the more advanced Perennial [Chajed 2022; Chajed et al. 2019, 2021] are separation logics capable of verifying crash-safety. In contrast to our work, Crash Hoare Logic and prior work using Perennial has only considered sequentially consistent memory and synchronously persisting writes without any weak behavior. When writes persist synchronously/atomically the content of durable storage is always in a single certain state. Therefore, rules for the post-crash modality include no non-determinism and are simpler "either/or" rules where some (volatile) resources are entirely lost at a crash and other (non-volatile) resources are preserved unchanged after a crash. In our setting, since the crash step is non-deterministic, the rules for the post-crash modality are significantly more involved. Consider for instance a rule such as PC-NA-MAPSTO which illustrates that the post-crash modality both introduces non-determinism (the quantified i), potentially takes resources away (represented both by $\langle \text{ifRec} \rangle$ and the lost states), and potentially adds new resources (the $\text{crashedIn}(\ell, \sigma_i)$).

Persistency Models. While our focus in this paper is on the logic, we remark on related work on persistency models. As mentioned, persistency models of the x86 and ARM architecture have been formalized [Khyzha and Lahav 2021; Raad et al. 2020b, 2019]. In parallel with our work, new variants of these that, like our semantics, are based on views have been presented [Cho et al. 2021]. It would be interesting to formally verify a correspondence between the explicit epoch persistency model and the x86 and ARM persistency models. We believe that our operational semantics could be used for this purpose. It would also be worthwhile to show an equivalence between our operational model and a model in a declarative or axiomatic style.

$$m \in \textsc{MEvent} ::= \mathsf{Al}_a(\ell, v) \mid \mathsf{R}_a(\ell, v) \mid \mathsf{W}_a(\ell, v) \mid \mathsf{RMW}(\ell, v_r, v_w) \mid \mathsf{RMW}_{\mathsf{fail}}(\ell, v) \mid \mathsf{FL}(\ell) \mid \mathsf{F} \mid \mathsf{FS}$$

$$\begin{aligned} \mathcal{V}, \mathcal{S}, \mathcal{F}, \mathcal{P}, \mathcal{B} &\in \textsc{View} \triangleq Loc \xrightarrow{\text{fin}} \mathbb{N} & \sigma &\in \textsc{Store} \triangleq Loc \xrightarrow{\text{fin}} \textsc{History} \\ \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle, \mathcal{T} &\in \textsc{ThreadView} \triangleq \textsc{View}^3 & h &\in \textsc{History} \triangleq \mathbb{N} \xrightarrow{\text{fin}} \textsc{Message} \\ \langle \sigma, \mathcal{P} \rangle, M &\in \textsc{MemConf} \triangleq \textsc{Store} \times \textsc{View} & \langle e, \mathcal{T} \rangle, t &\in \textsc{ThreadState} \triangleq \textsc{Exp} \times \textsc{ThreadView} \\ \langle v, \mathcal{S}_m, \mathcal{F}_m, \mathcal{P}_m \rangle &\in \textsc{Message} \triangleq \textsc{Val} \times \textsc{View}^3 & \langle M, \vec{t} \rangle, \rho &\in \textsc{MemConf} \times \textsc{List}(\textsc{ThreadState}) \end{aligned}$$

Fig. 13. Definitions of semantic objects used in the operational semantics

ACKNOWLEDGMENTS

We thank Hoang-Hai Dang for helpful conversations. We thank the anonymous reviewers for their comments that helped improve the presentation in the paper. We thank Amin Timany for useful suggestions during the mechanization effort. This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation. This work was co-funded by the European Union (ERC, CHORDS, 101096090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

A OPERATIONAL SEMANTICS

We define a small-step interleaving operational semantics for λ_{pmem}. The semantics formalizes the consistency and persistency models described informally in the paper.

To define how expressions interact with the memory we use two labeled transition systems (LTSs), one for expressions and one for the memory. This approach neatly keeps the memory model considerations separate from the rest of the language semantics. The labels for the LTSs are *memory events*, defined in Fig. 13, describing how expressions can interact with the memory.

A.1 Expression LTS

The LTS for expressions has the form $e \xrightarrow{m} e'; \vec{e}$, meaning that the expression e can step to e' with the label $m \in \textsc{MEvent} \cup \{\epsilon\}$ while forking the sequence of threads $\vec{e}$. The label ϵ is used for expressions that do not interact with the memory (pure reductions, *etc.*). A selection of expression transitions is seen in Fig. 14. First is the step for application (just to show that standard reductions work as expected), then the one for `fork` (the only rule where the sequence of forked threads is not the empty sequence ε), and then transitions that interact with the memory (*i.e.,* where $m \neq \epsilon$). Note how these transitions make it clear how the memory events correspond to operations in the language, for instance, the expression $\texttt{ref}_a\ v$ emits an event for allocation of the form $\mathsf{Al}_a(\ell, v)$, reading a value with $!_a\ \ell$ emits an event for reading $\mathsf{R}_a(\ell, v)$, and so on.

A.2 Memory LTS

We now wish to define an LTS for the memory. To this end, we need some semantic objects, defined in Fig. 13, which we now explain.

In a strong sequentially consistent memory threads always read the last write to a location, and hence the store (*i.e.,* the memory) can be modeled simply as a finite map from locations to values. In a weak persistent memory model, on the other hand, threads and recovery code may read out-of-date values. Therefore, the *store* is a finite map from locations to *histories*. Each history contains all writes to a location as a finite map from timestamps (natural numbers) to *messages*.

Every message corresponds to a write to the location and contains the written value and other data explained below. The timestamps correspond to the order of the writes.

Closely related to the definition of the store is the notion of a *view*: a finite map from locations to timestamps. Views are used to represent subsets of messages in the store. For a store σ, a view $\mathcal{V}$ intuitively represents all messages of the form $\sigma(\ell)(t)$ for $t \leq \mathcal{V}(\ell)$. We sometimes talk of the messages "in" a view to mean this set of messages. Views naturally form a semi-lattice where the least element $\bot$ is the empty partial function, where $\mathcal{V}_1 \sqsubseteq \mathcal{V}_2 \triangleq \forall \ell \in \mathsf{dom}(\mathcal{V}_1).\, \mathcal{V}_1(\ell) \leq \mathcal{V}_2(\ell)$, and where the least upper bound is given by $(\mathcal{V}_1 \sqcup \mathcal{V}_2)(\ell) \triangleq \max(\mathcal{V}_1(\ell), \mathcal{V}_2(\ell))$.

A *memory configuration* (MemConf) contains the entire state of the memory. It is a pair of a store and a view: $\langle \sigma, \mathcal{P} \rangle$. We refer to $\mathcal{P}$ as the *persist view*; it represents the messages in σ that are certain to have been persisted.

A *thread view* is a triple of views: $\langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle$. These views are a thread's *store view*, *flush view*, and *buffer view*. The store view $\mathcal{S}$ (flush view $\mathcal{F}$, respectively) is used to encode the weak memory order (persistent memory order). Messages in $\mathcal{S}$ are those that the thread knows of and that future memory operations will be ordered after. Messages in $\mathcal{F}$ are those that the thread knows have been flushed and fenced, meaning that will persist before any future memory operations by the thread. The buffer view $\mathcal{B}$ represents the messages that the thread has flushed.

A *message* is a tuple of the form: $\langle v, \mathcal{S}_m, \mathcal{F}_m, \mathcal{P}_m \rangle$. As mentioned, a message corresponds to a write and v is the value written. For an atomic write $\mathcal{S}_m$ and $\mathcal{F}_m$ is the writing thread's store view and flush view at the time of the write. An atomic read will acquire these views when reading the message. The $\mathcal{P}_m$ view enforces the persist order—a write can have persisted only if all messages in $\mathcal{P}_m$ have also persisted. In the operational semantics, the persist view of a message is only used in the reduction rule for a crash, corresponding to the fact that the persist order only affects crashes. For a message m, we write $m.v$, $m.\mathcal{S}$, *etc.* for its components.

The LTS for the memory has the form $\langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{m} \langle \sigma', \mathcal{P}' \rangle; \langle \mathcal{S}', \mathcal{F}', \mathcal{B}' \rangle$. As the outcome of a memory operation depends on the views of the thread making the operation, the LTS is parameterized both by a memory configuration and by a thread view: The transition rules appear in Fig. 14; to keep the presentation concise we make use of the notation

$$\lfloor \mathcal{V} \rfloor_{\mathsf{at}} \triangleq \mathcal{V} \qquad\qquad \mathcal{V}_0(\ell) = \begin{cases} \mathcal{V}(\ell) & \text{if } \ell \in \mathsf{dom}(\mathcal{V}) \\ 0 & \text{otherwise} \end{cases}$$
$$\lfloor \mathcal{V} \rfloor_{\mathsf{na}} \triangleq \bot$$

The $\lfloor \mathcal{V} \rfloor_a$ notation captures the effect of the access mode in several of the rules. For instance, the only difference between a write ($:=_{\mathsf{na}}$) and a release-write ($:=_{\mathsf{at}}$) is in which views are stored in the written message. We use $\mathcal{V}_0$ simply to be able to write $\mathcal{V}_0(\ell)$ even if ℓ is not certain to be in the domain of $\mathcal{V}$.

We now comment on the transition rules. The rule for allocation (with label $\mathsf{Al}_a(\ell, v)$) extends the store with a fresh location that contains a history with a single message at timestamp 0. In this rule and in the rule for writing the store view and flush view of the message is $\bot$ if the access mode is na. This is because non-atomic operations are not for synchronization between threads and therefore no views should be exchanged when performing them. In contrast, when the access mode is at then the thread's store view and flush view are included in the message. For a read with access mode at, rule (with label $\mathsf{R}_a(\ell, v)$) then merges the store view from the read message into the thread's store view. This ensures that the acquire-read actually acquires information from the thread whose write it is reading. The flush view from the message is only added to the thread's buffer view. The buffer view is never transferred between threads, it is only used within a thread to keep account of information that will be acquired at the next fence. Indeed, the buffer view is moved into a thread's flush view by the two fence rules (with labels F and FS).

Expression LTS

$$(\texttt{rec}\, f(x) = e)\; v \xrightarrow{\varepsilon} e[\texttt{rec}\, f(x) = e, v/f, x]; \varepsilon \qquad \texttt{fork}\,\{e\} \xrightarrow{\varepsilon} (); e \qquad \texttt{ref}_a\, v \xrightarrow{\mathsf{Al}_a(\ell,v)} \ell; \varepsilon$$

$$!_a\, \ell \xrightarrow{\mathsf{R}_a(\ell,v)} v; \varepsilon \qquad \ell :=_a v \xrightarrow{\mathsf{W}_a(\ell,v)} (); \varepsilon \qquad \texttt{flush}\, \ell \xrightarrow{\mathsf{FL}(\ell)} (); \varepsilon \qquad \texttt{fence} \xrightarrow{\mathsf{F}} (); \varepsilon$$

$$\texttt{fence}_{\texttt{sync}} \xrightarrow{\mathsf{FS}} (); \varepsilon \qquad \texttt{CAS}\, \ell\, v_1\, v_2 \xrightarrow{\mathsf{RMW}(\ell,v_1,v_2)} \texttt{true}; \varepsilon \qquad \frac{v_1 \neq v_l}{\texttt{CAS}\, \ell\, v_1\, v_2 \xrightarrow{\mathsf{RMW}_{\mathsf{fail}}(\ell,v_l)} \texttt{false}; \varepsilon}$$

Memory Model LTS

$$\frac{\ell \notin \mathrm{dom}(\sigma) \qquad h = \{0 \mapsto \langle v, \lfloor \mathcal{S} \rfloor_a, \lfloor \mathcal{F} \rfloor_a, \mathcal{F} \rangle\}}{\langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{\mathsf{Al}_a(\ell,v)} \langle \sigma[\ell \mapsto h], \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle} \qquad \frac{t = \mathcal{S}_0(\ell) \qquad \mathcal{B}' = \mathcal{B}[\ell \mapsto t]}{\langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{\mathsf{FL}(\ell)} \langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B}' \rangle}$$

$$\frac{\mathcal{S}_0(\ell) \leq t \qquad \sigma(\ell)(t) = \langle v, \mathcal{S}_m, \mathcal{F}_m, \_ \rangle}{\langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{\mathsf{R}_a(\ell,v)} \langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S} \sqcup \lfloor \mathcal{S}_m \rfloor_a, \mathcal{F}, \mathcal{B} \sqcup \lfloor \mathcal{F}_m \rfloor_a \rangle}$$

$$\frac{\mathcal{S}_0(\ell) < t \qquad \sigma(\ell) = h \qquad t \notin \mathrm{dom}(h) \qquad h' = h[t \mapsto \langle v, \lfloor \mathcal{S} \rfloor_a, \lfloor \mathcal{F} \rfloor_a, \mathcal{F} \rangle]}{\langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{\mathsf{W}_a(\ell,v)} \langle \sigma[\ell \mapsto h'], \mathcal{P} \rangle; \langle \mathcal{S}[\ell \mapsto t], \mathcal{F}, \mathcal{B} \rangle}$$

$$\frac{\begin{array}{c} \mathcal{S}_0(\ell) \leq t \qquad t+1 \notin \mathrm{dom}(h) \qquad \sigma(\ell) = h \\ h(t) = \langle v_m, \mathcal{S}_m, \mathcal{F}_m, \_ \rangle \qquad \mathcal{S}' = (\mathcal{S} \sqcup \mathcal{S}_m)[\ell \mapsto t+1] \qquad h' = h[t+1 \mapsto \langle v, \mathcal{S}', \mathcal{F} \sqcup \mathcal{F}_m, \mathcal{F} \sqcup \mathcal{F}_m \rangle] \end{array}}{\langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{\mathsf{RMW}(\ell,v,v')} \langle \sigma[\ell \mapsto h'], \mathcal{P} \rangle; \langle \mathcal{S}', \mathcal{F}, \mathcal{B} \sqcup \mathcal{F}_m \rangle}$$

$$\frac{\mathcal{S}_0(\ell) \leq t \qquad t+1 \notin \mathrm{dom}(h) \qquad \sigma(\ell) = h \qquad h(t) = \langle v_m, \mathcal{S}_m, \mathcal{F}_m, \_ \rangle}{\langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{\mathsf{RMW}_{\mathsf{fail}}(\ell,v)} \langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S} \sqcup \mathcal{S}_m, \mathcal{F}, \mathcal{B} \sqcup \mathcal{F}_m \rangle}$$

$$\langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{\mathsf{F}} \langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F} \sqcup \mathcal{B}, \mathcal{B} \rangle \qquad \langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{\mathsf{FS}} \langle \sigma; \mathcal{P} \sqcup \mathcal{B} \rangle; \langle \mathcal{S}, \mathcal{F} \sqcup \mathcal{B}, \mathcal{B} \rangle$$

Head Reduction

$$\frac{M; \mathcal{T} \xrightarrow{m} M'; \mathcal{T}' \qquad e \xrightarrow{m} e'; e_1 \ldots e_n}{M; \langle e, \mathcal{T} \rangle \to_{\mathsf{h}} M'; \langle e', \mathcal{T}' \rangle; \langle e_1, \mathcal{T}' \rangle \ldots \langle e_n, \mathcal{T}' \rangle} \qquad \frac{e \xrightarrow{\varepsilon} e'; e_1 \ldots e_n}{M; \langle e, \mathcal{T} \rangle \to_{\mathsf{h}} M; \langle e', \mathcal{T} \rangle; \langle e_1, \mathcal{T} \rangle \ldots \langle e_n, \mathcal{T} \rangle}$$

Thread-local and threadpool reduction

$$\frac{M; \langle e, \mathcal{T} \rangle \to_{\mathsf{h}} M'; \langle e', \mathcal{T}' \rangle; \vec{t}}{M; \langle K[e], \mathcal{T} \rangle \to_{\mathsf{t}} M'; \langle K[e'], \mathcal{T}' \rangle; \vec{t}} \qquad \frac{M; t \to_{\mathsf{t}} M'; t'; \vec{t}}{\langle M; \vec{t}_l t \vec{t}_r \rangle \to_{\mathsf{tp}} \langle M'; \vec{t}_l t' \vec{t}_r \vec{t} \rangle}$$

Fig. 14. Expression and Memory Model LTS transitions

Due to the condition $\mathcal{S}_0(\ell) \leq t$ in rule for reading, a read may non-deterministically read any message for ℓ with a timestamp greater than the thread's timestamp in its store view for ℓ.

Note that the rule for writing ensures that the thread's flush view $\mathcal{F}$ is transferred irrespectively of access mode; this is to ensure that the persist order is recorded (it is used when we account for crashes, in Appendix A.4).

The rules for flushes and fences are rather straightforward. Flushing a location moves the thread's timestamp for the location in its store view into its buffer view. The two rules for fences, move a thread's buffer view into its flush view. The rule for the synchronous fence additionally moves the buffer into the memory configuration's persist view as well.

A.3 Machine Reductions

We define a head reduction $\rightarrow_{\mathrm{h}}$ for a memory and a thread state (a pair of an expression and a thread view) by combining the LTSs for the memory and for expressions. It is given by the two rules seen in Fig. 14. The first rule is for expression steps that interact with the memory and the second for those that do not.

As is standard we use evaluation contexts K to lift the head reduction to a per-thread reduction $\rightarrow_{\mathrm{t}}$ which again is lifted to a threadpool reduction $\rightarrow_{\mathrm{tp}}$ that non-deterministically picks a thread from the threadpool to reduce. They are each given by a single rule seen in Fig. 14. Here K is an evaluation context; the definition of evaluation contexts for λ_{pmem} is entirely standard, capturing a call-by-value left-to-right evaluation order, and has thus been omitted.

A.4 Accounting for Crashes

The state of the memory after a crash is determined by a *crash view* C. The crash view represents all the messages that persisted before the crash. It has to be a *consistent cut* in the sense that no message can have persisted without all of the messages in its persist view also having persisted:

Definition A.1. A view C is a *consistent cut* of a store σ, written $\mathrm{consistent}(\sigma, C)$, iff for every $C(\ell) = t$ there exists a history h such that $\sigma(\ell) = h$ and $t \in \mathrm{dom}(h)$. Furthermore, for all $t' \in \mathrm{dom}(h)$ where $t' \leq t$ there exists a message m such that $h(t') = m$ and $m.\mathcal{P} \sqsubseteq C$.

With this in hand, we can define the *crash step* reduction $\xrightarrow{\lightning}$. This reduction goes from memory configurations to memory configurations and describes what can happen to the memory at a crash. It is generated by a single rule:

M-CRASH
$$\frac{\begin{array}{cc} & \mathcal{P} \sqsubseteq C \qquad \mathrm{consistent}(\sigma, C) \\ \mathrm{dom}(\sigma') = \mathrm{dom}(C) & \forall \ell \in \mathrm{dom}(C).\, \sigma'(\ell) = \{0 \mapsto \langle \sigma(\ell)(C(\ell)).v, \bot, \bot, \bot \rangle\} \end{array}}{\langle \sigma, \mathcal{P} \rangle \xrightarrow{\lightning} \langle \sigma', \mathrm{viewToZero}(C) \rangle}$$

The first two assumptions ensure that C is a consistent cut and that it includes all the definitely persisted messages in $\mathcal{P}$. The next line serves to constrict the new store σ', created by picking out a message for each recovered location. The new persist view $\mathrm{viewToZero}(C)$ is the view such that $\mathrm{viewToZero}(C)(\ell) = 0$ for all $\ell \in \mathrm{dom}(C)$ and which is undefined for all $\ell \notin \mathrm{dom}(C)$. That is, the crash view but with 0 at every entry.

Finally, we define a *recoverable execution relation* $\Rightarrow_{\mathrm{r}}$, which expresses what it means to execute a program together with a recovery program e_r that is run after each crash. The relation has the form $e_r; \rho \Rightarrow_{\mathrm{r}} \rho'; s$ where e_r is the recovery expression, ρ and ρ' are machine configurations, and $s \in \{\mathrm{NotCrashed}, \mathrm{Crashed}\}$ is a crash-status indicating whether the execution has been crash free or has crashed along the way.

REXEC-NORMAL
$$\frac{\rho \rightarrow^{*}_{\mathrm{tp}} \rho'}{e_r; \rho \Rightarrow_{\mathrm{r}} \rho'; \mathrm{NoCrash}}$$

REXEC-CRASHED
$$\frac{\rho \rightarrow^{*}_{\mathrm{tp}} \langle M, \vec{t} \rangle \qquad M \xrightarrow{\lightning} M' \qquad e_r; \langle M', [\langle e_r, \langle \bot, \bot, \bot \rangle \rangle] \rangle \Rightarrow_{\mathrm{r}} \rho'; s}{e_r; \rho \Rightarrow_{\mathrm{r}} \rho'; \mathrm{Crashed}}$$

Note that in contrast to the other relations we have defined above this is a *big step* relation. The first rule says that a machine can execute (without crashes) per the threadpool reduction with the label NoCrash. The second rule says that a machine may execute normally for some number of steps (the first assumption), then let the memory take a crash step (the second assumption), and then keep executing with the memory after the crash and a single thread executing the recovery expression. Note that at a crash all the running threads $\vec{t}$ are discarded.

B BASESPIREA – THE LOW-LEVEL LOGIC

In this section we present BaseSpirea, a program logic for λ_{pmem}. The logic is built on top of Iris and Perennial by instantiating the Perennial program logic framework. Before we proceed, we briefly explain the relationship between Iris and Perennial and what such an instantiation entails.

B.1 Instantiating Perennial

Iris includes both a base logic and a program logic framework. The base logic contains the fundamental features of Iris, such as the separation logic connectives and ghost state, but not program verification capabilities. Perennial, in turn, builds a program logic framework on top of the Iris base logic. Unlike the program logic in Iris, Perennial's is able to reason about crashes and to verify crash-safety, through a combination of powerful features made for this purpose. Here "framework" describes the fact that one can instantiate the program logic with any suitable programming language. We instantiate the framework with λ_{pmem}, which, by no coincidence, is one such suitable language. By doing this instantiation one gets the basic building blocks of a program logic "for free". These building blocks include the definition of Hoare triples (and related notions explained in the next section) and language independent structural rules for working with them. The instantiator (*i.e.,* us) then defines language specific assertions and proof rules that must be proven sound. This is done in tandem with a so-called *state interpretation* that is picked as part of the instantiation. The state interpretation's purpose is to link the physical state (*i.e.,* the state in the operational semantics) with logical ghost state. At this level, our state interpretation is fairly standard and we do not give the details here—the interested reader can find them in our Coq mechanization. One noteworthy aspect of our state interpretation is that it is parameterized over an "extra" resource. In BaseSpirea this extra resource is simply True, but when building Spirea on top of BaseSpirea, we use it to inject additional resources into the state interpretation.

We next describe the most important features of the Perennial program logic that we inherit (Appendix B.2); the language specific assertions BaseSpirea adds (Appendix B.3); some of the program rules that BaseSpirea includes (Appendix B.4); and finally we give the adequacy result of the logic (Appendix B.5).

B.2 The Perennial Program Logic

In this section we explain the most important features from Perennial that are also present in BaseSpirea. Note that in this subsection we use e to denote expressions from the point of view of Perennial. When instantiated with λ_{pmem}, such an expression is in fact a thread state (defined in Appendix A).

In addition to the well-known Hoare triple, Perennial includes a *crash Hoare triple*[8] of the form $\{P\}\, e\, \{Q\}\{Q_c\}$. Here P and Q are standard pre- and postconditions and the fourth component Q_c is a *crash condition* that must hold during every step of execution of e. Since Q_c holds at every step, if a crash occurs at some point, then Q_c will necessarily hold at that point. Hence, the crash-condition is a property that recovery code can rely on after a crash.

In addition to standard language independent structural rules (a frame rule, a bind rule, *etc.*), the key rule for deriving a crash Hoare triple is Htc-atomic seen in Fig. 15

The rule states that to prove a crash Hoare triple for an atomic expression e, it suffices to prove that the pre-condition implies Q_c and an ordinary Hoare triple for e holds with Q_c added to the postcondition. Since e is atomic and can take only a single step, it suffices to show the crash condition before and after this single step. Note the use of the standard (non-separating) conjunction $\wedge$. This makes it possible to use all the resources one has at hand to show both Q and

[8]Really, it is a quadruple, but we stick with the Hoare triple terminology

Q_c. This is a crucial aspect of crash conditions: they can be established without losing the resources necessary to show them.[9] The use of $\wedge$ is sound since when the program runs it will *either* take a normal step of execution (in which case the proof of Q is needed) *or* crash (in which case the proof of Q_c is needed). Since both cannot happen at the same time, it is not necessary to show the two conjuncts for disjoint resources. The Htc-atomic rule is important since it, in combination with the structural rules, allows us to show a crash Hoare triple by showing a normal Hoare triple. This explains why we show rules for normal Hoare triples later on in this section.

To reason about the combination of a program e and its associated recovery program e_r, Perennial offers a *recovery Hoare triple*[10] of the form $\{P\}\ e \circlearrowleft e_r\ \{Q\}\{Q_r\}$. The intuitive reading is: given that P holds initially, it is safe to execute e with the recovery program e_r. If e terminates in a value v without crashing then $Q(v)$ holds. If, on the other hand, one or more crashes occur during execution (of e and e_r) then, if e_r terminates in a value v, then $Q_r(v)$ holds. Beware that the Q_r plays a different role from the Q_c used in a crash Hoare triple.

A *post-crash modality* $\langle \mathrm{PC} \rangle$ internalises in the logic how resources are affected by a crash. The assertion $\langle \mathrm{PC} \rangle\ P$ means that P holds after a crash and a rule of the form $P \vdash \langle \mathrm{PC} \rangle\ Q$ means that if one has P then one has Q after a crash.

Since the post-crash modality depends intrinsically on the semantics of the specific programming language one reasons about, we do not get the post-crash modality by instantiating Perennial; we have to define it ourselves. The *idea* of a post-crash modality, however, is from Perennial. That being said, ours is more complicated than earlier ones used with Perennial due to the more intricate semantics for crashes in λ_{pmem} (for more details, see the discussion of related work in §7).

Per the idempotence rule Htr-idempotence one can show a recovery Hoare triple for a program e and recovery program e_r by showing a crash Hoare triple for e and one for e_r. In both cases the crash condition is Q_r, such that e_r can rely on this resource; not directly though, as the crash itself might change Q_r, hence the inclusion of the post-crash modality. Since e_r itself maintains the crash condition Q_r, any number of crashes during e_r are still safe.

The Perennial program logic contains other features. For instance, *crash borrows* that make it possible to transfer and split crash conditions between threads. But what we have explained thus far suffices for this paper, so we proceed to explain the assertions specific to BaseSpirea.

B.3 Assertions in BaseSpirea

Since the store in our operational semantics contains not just single values, but entire histories, the points-to predicate in BaseSpirea $\ell \hookrightarrow_{\mathsf{h}} h$ naturally associates a location with a history h. Except for this, it is similar to the normal separation logic points-to predicate.

The assertion $\mathrm{valid}(\mathcal{S})$ states that a view $\mathcal{S}$ is valid. This means that if $\mathcal{S}(\ell) = t$ then the history for ℓ in the physical store actually contains a message with at least the timestamp t. Knowing this is necessary to conclude, for instance, that performing a read with the store view $\mathcal{S}$ is safe.

The assertion $\mathrm{persisted}(\mathcal{P})$ means that the view $\mathcal{P}$ is included in the persist view in the physical state. It does not entail any ownership (in the separation logic sense) of the physical persist view. Since it only expresses a lower bound and since the physical persist view only grows during normal execution it is persistent.[11]

The assertion $\mathrm{crashedAt}(C)$ means that at *the last crash* the consistent cut that the machine crashed at was C. The assertion $\mathrm{crashedAt}(C)$ is persistent and has agreement (crashed-at-agree).

[9] This is in contrast to normal Iris invariants, where one has to sacrifice ownership of the resources necessary to show the invariant.

[10] Again, we stick with the Hoare triple terminology, even if more than three components are involved.

[11] Not to be confused with persistent memory, in Iris a persistent proposition is one that does not entail exclusive ownership but only represents duplicable knowledge. $\square\, P$ means that P always holds and a proposition P is persistent if $P \vdash \square\, P$.

Htc-atomic

$$\frac{\text{atomic}(e) \qquad P \twoheadrightarrow Q_c}{\{P\}\, e\, \{Q \land Q_c\} \vdash \{P\}\, e\, \{Q\}\{Q_c\}}$$

Htr-idempotence

$$\frac{\{P\}\, e\, \{Q\}\{Q_r\} \qquad Q_r \twoheadrightarrow \langle \mathrm{PC} \rangle\, R \qquad \{R\}\, e_r\, \{Q_r\}\{Q_r\}}{\{P\}\, e \circlearrowleft e_r\, \{Q\}\{Q_r\}\,.}$$

crashed-at-agree

$$\mathrm{crashedAt}(C) * \mathrm{crashedAt}(C') \vdash C = C'$$

persisted-sep

$$\mathrm{persisted}\,\mathcal{P}_1 * \mathrm{persisted}\,\mathcal{P}_2 \dashv\vdash \mathrm{persisted}(\mathcal{P}_1 \sqcup \mathcal{P}_2)$$

Pc-persisted

$$\mathrm{persisted}(\mathcal{P}) \vdash \langle \mathrm{PC} \rangle\, \mathrm{persisted}(\mathrm{viewToZero}(\mathcal{P})) * \exists C \sqsupseteq \mathcal{P}.\ \mathrm{crashedAt}(C)$$

Pc-points-to

$$\ell \hookrightarrow_{\mathrm{h}} h \vdash \langle \mathrm{PC} \rangle\, \exists C. \mathrm{crashedAt}(C) * \left(\ell \notin \mathrm{dom}(C) \lor \left(\exists t, m. \begin{array}{l} h(t) = m * C(\ell) = t * m.\mathcal{P} \sqsubseteq C * \\ \ell \hookrightarrow_h \{0 \mapsto \langle m.v, \bot, \bot, \bot \rangle\} \end{array} \right) \right)$$

Fig. 15. Selected rules for assertions in BaseSpirea

We now need to describe how the assertions interact with the post-crash modality. There are no rules for valid or crashedAt. These resource do not imply any non-trivial resources after a crash, *i.e.*,, the assertions valid($\mathcal{S}$) and crashedAt(C) are lost under the post-crash modality.

For persisted($\mathcal{P}$) we have the rule Pc-persisted. It says that given persisted($\mathcal{P}$), then after a crash persisted holds for the same view but with zero at every entry. Furthermore, there exists some view C such that $C \sqsupseteq \mathcal{P}$ and crashedAt(C) holds. This rule is sound because persisted($\mathcal{P}$) is a lower bound on the persist view in the physical state and a crash view has to include the persist view (recall the rule M-crash in the operational semantics).

The rule Pc-points-to for the points-to predicate is a bit more involved. After a crash, we again have crashedAt(C) for some C and two distinct cases: the location was either lost or recovered at the crash. In the first case, the location must not be present in the crash view, $\ell \notin \mathrm{dom}(C)$, and we have, of course, lost the points-to predicate. In the latter case, $C(\ell) = t$ for some t, the message $h(t)$ was recovered, and we now have a points-to predicate for the recovered message. Furthermore, the view $\mathcal{P}$ in the message must be included in C. This internalizes the fact that C must be a consistent cut and hence respect $\mathcal{P}$ in the message. This is what makes it possible to do the kind of "backwards reasoning" for recovery code that we discussed earlier.

Let us see how the post-crash rules for persisted($\mathcal{P}$) and $\ell \hookrightarrow_{\mathrm{h}} h$ work together if we have both before a crash. Since crashedAt has agreement (crashed-at-agree) the two crash views gained by Pc-persisted and Pc-points-to must be equal. Hence, if one knows that $\ell \in \mathrm{dom}(\mathcal{P})$ then one can rule out the first case in the disjunction in Pc-points-to and obtain a points-to predicate after the crash.

B.4 BaseSpirea Program Logic

BaseSpirea includes proof rules for all programming language constructs of λ_{pmem}. The rules for the memory-related operations are shown in Fig. 16; we only include these rules as the proof rules for the remaining part of λ_{pmem} are as in standard Iris, see, *e.g.*, [Jung et al. 2018] (but, we hasten to point out, we have also proven those standard rules sound!).

The memory-related rules very closely reflect the underlying operational semantics and thus we do not explain them in great detail, but only make a few general observations. Since the state of a thread in our operational semantics is described not only by an expression but also by a thread view, the "program" in our Hoare-triple is a thread state and not just an expression. All the rules that involve reading and writing to a location include valid($\mathcal{S}$) in their precondition and valid($\mathcal{S}'$) in their postcondition. The knowledge of validity is necessary to conclude that reads do not get stuck, as mentioned above.

Ht-alloc
$$\{\mathrm{valid}(\mathcal{S})\}\ \mathbf{ref}_a\ v; \langle \mathcal{S}, \mathcal{F}, \mathcal{B}\rangle\ \left\{\ell; \langle \mathcal{S}', \mathcal{F}', \mathcal{B}'\rangle.\ \ell \hookrightarrow_{\mathsf{h}} \{0 \mapsto \langle v, \lfloor \mathcal{S} \rfloor_a, \lfloor \mathcal{F} \rfloor_a, \mathcal{F}\rangle\} * \mathcal{S} = \mathcal{S}' * \mathcal{F} = \mathcal{F}' * \mathcal{B} = \mathcal{B}'\right\}$$

Ht-read
$$\left\{\begin{matrix}\mathrm{valid}(\mathcal{S}) * \\ \ell \hookrightarrow_{\mathsf{h}} h\end{matrix}\right\} !_a\,\ell; \langle \mathcal{S}, \mathcal{F}, \mathcal{B}\rangle \left\{\begin{matrix} v; \langle \mathcal{S}', \mathcal{F}', \mathcal{B}'\rangle.\ \exists t.\ \mathcal{S}_0(\ell) \le t * h(t) = \langle v_m, \mathcal{S}_m, \mathcal{P}_m, \_\rangle * v = v_m * \\ \mathcal{S}' = \mathcal{S} \sqcup \lfloor \mathcal{S}_m \rfloor_a * \mathcal{F}' = \mathcal{F} * \mathcal{B} = \mathcal{B}' \sqcup \lfloor \mathcal{P}_m \rfloor_a * \mathrm{valid}(\mathcal{S}') * \ell \hookrightarrow_{\mathsf{h}} h\end{matrix}\right\}$$

Ht-store
$$\left\{\begin{matrix}\mathrm{valid}(\mathcal{S}) * \\ \ell \hookrightarrow_{\mathsf{h}} h\end{matrix}\right\} \ell :=_a v; \langle \mathcal{S}, \mathcal{F}, \mathcal{B}\rangle \left\{\begin{matrix} w; \langle \mathcal{S}', \mathcal{F}', \mathcal{B}'\rangle.\ \exists t.\ \mathcal{S}_0(\ell) < t * t \notin \mathrm{dom}(h) * w = () * \mathcal{S}' = \mathcal{S}[\ell \mapsto t] * \\ \mathcal{F}' = \mathcal{F} * \mathcal{B}' = \mathcal{B} * \mathrm{valid}(\mathcal{S}') * \ell \hookrightarrow_{\mathsf{h}} h[t \mapsto \langle v, \lfloor \mathcal{S}' \rfloor_a, \lfloor \mathcal{F} \rfloor_a, \mathcal{F}\rangle]\end{matrix}\right\}$$

Ht-flush
$$\{\ell \hookrightarrow_{\mathsf{h}} h\}\ \mathbf{flush}\ \ell; \langle \mathcal{S}, \mathcal{F}, \mathcal{B}\rangle\ \left\{w; \langle \mathcal{S}', \mathcal{F}', \mathcal{B}'\rangle.\ \mathcal{S}' = \mathcal{S} * \mathcal{F}' = \mathcal{F} * \mathcal{B}' = \mathcal{B}[\ell \mapsto \mathcal{S}_0(t)] * \ell \hookrightarrow_{\mathsf{h}} h\right\}$$

Ht-fence
$$\{\mathrm{True}\}\ \mathbf{fence}; \langle \mathcal{S}, \mathcal{F}, \mathcal{B}\rangle\ \left\{w; \langle \mathcal{S}', \mathcal{F}', \mathcal{B}'\rangle.\ \mathcal{S}' = \mathcal{S} * \mathcal{F}' = \mathcal{F} \sqcup \mathcal{B} * \mathcal{B}' = \mathcal{B}\right\}$$

Ht-fence-sync
$$\{\mathrm{True}\}\ \mathbf{fence_{sync}}; \langle \mathcal{S}, \mathcal{F}, \mathcal{B}\rangle\ \left\{w; \langle \mathcal{S}', \mathcal{F}', \mathcal{B}'\rangle.\ \mathcal{S}' = \mathcal{S} * \mathcal{F}' = \mathcal{F} \sqcup \mathcal{B} * \mathcal{B}' = \mathcal{B} * \mathrm{persisted}(\mathcal{B})\right\}$$

Fig. 16. Selected rules for Hoare triples in BaseSpirea

The rules for flushing and fences are very simple. The rule Ht-flush requires a points-to predicate only to ensure that the flushed location actually exists in the store. The only difference between Ht-fence and Ht-fence-sync is that the later includes $\mathrm{persisted}(\mathcal{B})$ in the postcondition. The rule for a synchronous fence is the only way to get the $\mathrm{persisted}(P)$ assertion.

B.5 Soundness

The soundness theorem for BaseSpirea states that a recovery Hoare triple for a program proven inside the logic implies a safety result about the program with respect to the operational semantics—independently of the logic. Since this result is the same in the soundness theorem for both BaseSpirea and Spirea we define it separately, such that we can reuse it in both theorems.

Definition B.1. For expressions e and e_r, memory configuration M, and meta-level predicates on values Φ and Φ_r, we say that $\mathrm{safe}(e, e_r, M, \Phi, \Phi_r)$ holds if, for any recoverable execution
$$e_r; \langle M, [\langle e, \langle \bot, \bot, \bot\rangle\rangle]\rangle \Rightarrow_{\mathsf{r}} \langle M, \vec{t}\rangle; s$$
it is the case that: (1) For every thread $\langle e, \mathcal{T}\rangle \in \vec{t}$, if e is not a value then the thread is not stuck. (2) For $\langle e', \mathcal{T}\rangle = (\vec{t})_1$, if e' is a value v (*i.e.*, the initial expression e terminated) then $\Phi(v)$ holds if $s = \mathrm{NotCrashed}$ and $\Phi_r(v)$ holds if $s = \mathrm{Crashed}$.

With this safety definition we state the soundness theorem.

Theorem B.2 (soundness). *Let e, e_r, $\langle \sigma, \mathcal{P}\rangle$, Φ, and Φ_r be as in Def. B.1. If the following recovery Hoare triple is provable in BaseSpirea*
$$\left\{\mathrm{valid}(\mathcal{P}) * \mathop{\Huge *}_{\ell \in \mathrm{dom}(\sigma)} \ell \hookrightarrow_h \sigma(\ell)\right\} \langle e, \mathcal{P}\rangle \circlearrowleft \langle e_r, \bot, \bot, \bot\rangle\ \{\Phi\}\{\Phi_r\}$$
then $\mathrm{safe}(e, e_r, \langle \sigma, \mathcal{P}\rangle, \Phi, \Phi_r)$ *holds.*

Note that the theorem applies to a store and persist view that is not necessarily empty to begin with. Importantly, this makes it possible to apply it to programs that assume already existing and persisted locations.

```
$\{x \hookrightarrow_{\mathsf{na}} [0] * y \hookrightarrow_{\mathsf{at}} 0 * z \hookrightarrow_{\mathsf{na}} 0 * x \gtrsim_{\mathsf{p}} 0 * z \gtrsim_{\mathsf{p}} 0\}$

$\{x \hookrightarrow_{\mathsf{na}} [0] * y \hookrightarrow_{\mathsf{at}} 0\}$
$x :=_{\mathsf{na}} 1$;
$\{x \hookrightarrow_{\mathsf{na}} [0, 1]\}$
flush $x$;
$\{x \hookrightarrow_{\mathsf{na}} [0, 1] * \langle \mathsf{PF} \rangle\, x \gtrsim_{\mathsf{f}} 1\}$
fence;
$\{x \hookrightarrow_{\mathsf{na}} [0, 1] * x \gtrsim_{\mathsf{f}} 1\}$
$y :=_{\mathsf{at}} 1$
$\{x \hookrightarrow_{\mathsf{na}} [0, 1] * y \hookrightarrow_{\mathsf{at}} 1\}$

||

$\{y \hookrightarrow_{\mathsf{at}} 0 * z \hookrightarrow_{\mathsf{na}} 0\}$
if $!_{\mathsf{at}}\, y = 1$
then
  $\{z \hookrightarrow_{\mathsf{na}} 0 * \langle \mathsf{PF} \rangle\, x \gtrsim_{\mathsf{f}} 1\}$
  fence;
  $\{z \hookrightarrow_{\mathsf{na}} 0 * x \gtrsim_{\mathsf{f}} 1\}$
  $z :=_{\mathsf{na}} 1$
  $\{\mathrm{True}\}$
else
  $\{\mathrm{True}\}()\ \{\mathrm{True}\}$

$\{x \hookrightarrow_{\mathsf{at}} n * z \hookrightarrow_{\mathsf{na}} m\}$
if $!_{\mathsf{na}}\, z = 1$
then
  $\{x \hookrightarrow_{\mathsf{at}} n * z \hookrightarrow_{\mathsf{na}} 1 * x \gtrsim_{\mathsf{f}} 1\}$
  $\{x \hookrightarrow_{\mathsf{at}} 1 * z \hookrightarrow_{\mathsf{na}} 1\}$
  assert $!_{\mathsf{na}}\, x = 1$
  $\{\mathrm{True}\}$
else
  $\{\mathrm{True}\}()\ \{\mathrm{True}\}$
```

Fig. 17. Proof outline for the durable message passing example

C DURABLE MESSAGE PASSING

To demonstrate resource transfer between threads we verify the durable message passing example from Fig. 2b. We include recovery code, the safety of which depends on the property that the example satisfies. The recovery code is seen in the proof outline in Fig. 17.

For all locations we pick the set of abstract states $\{0, 1\}$ and we choose the ψ in the invariant to be the identify function. The invariants are:

$$\phi_x(n, v) \triangleq n = v \qquad \phi_y(n, v) \triangleq (n = v = 0 * \mathrm{tok}) \vee (n = v = 1 * x \gtrsim_{\mathsf{f}} 1) \qquad \phi_z(n, v) \triangleq (n = v = 0) \vee (n = v = 1 * x \gtrsim_{\mathsf{f}} 1)$$

Here "tok" is an exclusive token implemented using standard Iris ghost state. By exclusive we mean that "tok $*$ tok" is a contradiction. We choose the following crash condition:

$$\langle \mathsf{PC} \rangle\, \exists n, m.\, x \gtrsim_{\mathsf{p}} n * x \hookrightarrow_{\mathsf{na}} n * z \gtrsim_{\mathsf{p}} m * z \hookrightarrow_{\mathsf{na}} m$$

The crash condition does not mention y as the recovery code does not use y. The crash condition is easy to show at every step. Since the example involves two threads we need to mention that, using features derived from Perennial, it is possible to split a crash condition between threads where each thread is responsible for maintaining its part. This means that we can use standard concurrent-separation-logic-style thread-local reasoning and prove each of the two threads separately.

The key idea of the proof that we want to highlight is at the write $y :=_{\mathsf{at}} 1$ in the left thread. Here we use Ht-at-write where σ is 0 and σ_t is 1. Showing the invariant is trivial, the tricky part is the last conjunct in the precondition of Ht-at-write, namely that the written state fits in the ordered history of states. To do this we assume some σ_c that is either 0 or 1 (here we crucially use that the abstract state contains only 0 and 1). The latter case is trivial, so suppose $\sigma_c = 0$. In the first case we have the invariant for 0 twice. Since $\phi(0, v)$ contains an exclusive token this is a contradiction. The use of the exclusive token for the state 0 in ϕ_y ensures that the abstract history can only contain 0 once. Hence, when writing the state 1 we can rule out the case where another thread writes 0 that ends up succeeding our write of 1 (which would violate the order of the abstract history). Notice how the argument is modular, we do not assume any knowledge of any other threads, only that the invariant is respected.

We remark that Bila et al. [2022] verified a variant of the durable message passing example where the flush and fence for x is moved from the left to the right thread. We have verified this variant as well in Coq and in this example the ψ function in the invariant for y is *not* the identity function.

```
$\{\mathrm{isLock}(\mathrm{lk}, P_{lk}, P_{c,lk}) * z \hookrightarrow_{\mathsf{na}} [\mathbf{false}] * z \gtrsim_{\mathsf{p}} \mathbf{false}\}$

$\{\mathrm{True}\}$
acquire lk
$\left\{\begin{matrix} x \gtrsim_{\mathsf{p}} \mathbf{false} * x \gtrsim_{\mathsf{f}} b * x \hookrightarrow_{\mathsf{na}} \vec{\sigma} \mathbin{+\!\!+} [b] * \\ y \gtrsim_{\mathsf{p}} \mathbf{false} * y \gtrsim_{\mathsf{f}} b * y \hookrightarrow_{\mathsf{na}} \vec{\sigma} \mathbin{+\!\!+} [b] \end{matrix}\right\}$
$x :=_{\mathsf{na}} \mathbf{true}$;
$\{x \hookrightarrow_{\mathsf{na}} \sigma \mathbin{+\!\!+} [b, \mathbf{true}]\}$
$y :=_{\mathsf{na}} \mathbf{true}$;
$\{y \hookrightarrow_{\mathsf{na}} \sigma \mathbin{+\!\!+} [b, \mathbf{true}]\}$
flush $x$;
$\{\langle \mathsf{PF} \rangle\, x \gtrsim_{\mathsf{f}} \mathbf{true}\}$
flush $y$;
$\{\langle \mathsf{PF} \rangle (x \gtrsim_{\mathsf{f}} \mathbf{true} * y \gtrsim_{\mathsf{f}} \mathbf{true})\}$
fence;
$\{x \gtrsim_{\mathsf{f}} \mathbf{true} * y \gtrsim_{\mathsf{f}} \mathbf{true}\}$
release lk
$\{\mathrm{True}\}$

||

$\{z \hookrightarrow_{\mathsf{na}} [\mathbf{false}]\}$
acquire lk
$\left\{\begin{matrix} x \gtrsim_{\mathsf{p}} \mathbf{false} * x \gtrsim_{\mathsf{f}} b * x \hookrightarrow_{\mathsf{na}} \vec{\sigma} \mathbin{+\!\!+} [b] * \\ y \gtrsim_{\mathsf{p}} \mathbf{false} * y \gtrsim_{\mathsf{f}} b * y \hookrightarrow_{\mathsf{na}} \vec{\sigma} \mathbin{+\!\!+} [b] \end{matrix}\right\}$
if $!_{\mathsf{na}}\, x = \mathbf{true}$
then
  $\{b = \mathbf{true}\}$
  $\left\{\begin{matrix} x \gtrsim_{\mathsf{f}} \mathbf{true} * x \hookrightarrow_{\mathsf{na}} \vec{\sigma} \mathbin{+\!\!+} [\mathbf{true}] * \\ y \gtrsim_{\mathsf{f}} \mathbf{true} * y \hookrightarrow_{\mathsf{na}} \vec{\sigma} \mathbin{+\!\!+} [\mathbf{true}] \end{matrix}\right\}$
  $z :=_{\mathsf{na}} 1$
  $\{\mathrm{True}\}$
else ()
$\{\mathrm{True}\}$
release lk
$\{\mathrm{True}\}$

$\left\{\begin{matrix} x \gtrsim_{\mathsf{p}} b_x * x \hookrightarrow_{\mathsf{na}} \vec{\sigma} \mathbin{+\!\!+} [b_x] * \\ y \gtrsim_{\mathsf{p}} b_y * y \hookrightarrow_{\mathsf{na}} \vec{\sigma} \mathbin{+\!\!+} [b_y] \\ z \gtrsim_{\mathsf{p}} b_z * z \hookrightarrow_{\mathsf{na}} \vec{\sigma} \mathbin{+\!\!+} [b_z] \end{matrix}\right\}$
if $!_{\mathsf{na}}\, z = \mathbf{true}$
then
  $\{x \gtrsim_{\mathsf{f}} \mathbf{true} * y \gtrsim_{\mathsf{f}} \mathbf{true}\}$
  $\left\{\begin{matrix} x \hookrightarrow_{\mathsf{na}} \vec{\sigma} \mathbin{+\!\!+} [\mathbf{true}] * \\ y \hookrightarrow_{\mathsf{na}} \vec{\sigma} \mathbin{+\!\!+} [\mathbf{true}] \end{matrix}\right\}$
  assert $!_{\mathsf{na}}\, x = \mathbf{true}$
  assert $!_{\mathsf{na}}\, y = \mathbf{true}$
  $\{\mathrm{True}\}$
else
  $\{\mathrm{True}\}$() $\{\mathrm{True}\}$
```

Fig. 18. Proof outline for the atomic persists example. At the top is the pre-crash code and below the recovery code

Since the message is sent before anything is flushed the information in the message is lost at a crash, and hence the state needs to change at a crash. For the details see our Coq mechanization.

D ATOMIC PERSISTS EXAMPLE

We explain the proof of the atomic persist example. The program can be seen in the proof outline in Fig. 18. The two threads use a shared lock that they both attempt to acquire. When the left thread acquires the lock it writes **true** to the locations x and y. It then flushes both locations and carries out a fence before it releases the lock. When the right thread acquires the lock it reads x and if it reads **true** it writes **true** to z.

The property that we want to show is that the write to z must be ordered after the two writes to x and y. In other words if the right thread reads **true** from x it must also be the case that it would read **true** from y if it where to do so. So, due to the use of the lock the two separate writes performed by the left thread appear as one atomic write to the right thread, *i.e.,* the right thread either sees none of the writes or all of the writes.

For the locations x and y we use a simple invariant with an abstract state of booleans $\{\texttt{true}, \texttt{false}\}$ and $\phi_b(\sigma, b) \triangleq \sigma = b$ as the invariant.

To verify the example we prove a crash-aware lock specification for a (volatile) lock using BaseSpirea. The specification that we show for the lock is identical to the crash-aware lock specification proposed by Chajed [2022, Chapter 3] (note that while the specification is the same our proof is different as it has to account for weak persistent memory whereas Perennial's lock assumes sequentially consistent memory). Since our specification is identical to their we do not repeat it here. The key point that is relevant to our present goal is that the assertion for the lock has the form $\mathrm{isLock}(v, P, P_c)$ and means that the lock protects both a resource P, as the standard CSL lock specification, and a *crash-resource* P_c, which essentially is a crash condition that the lock guarantees to preserve.

We want the lock to own the points-to predicates for x and y. Furthermore, the resource should state that x and y have the same last state and that that state has been flushed. Finally, they should be persisted in at least the initial state.

$$\begin{aligned} P_{lock} \triangleq \exists \vec{\sigma}, b.\, & x \gtrsim_{\mathsf{p}} \texttt{false} * x \gtrsim_{\mathsf{f}} b * x \hookrightarrow_{\mathsf{na}} \vec{\sigma} \mathbin{+\!\!+} [b] * \\ & y \gtrsim_{\mathsf{p}} \texttt{false} * y \gtrsim_{\mathsf{f}} b * y \hookrightarrow_{\mathsf{na}} \vec{\sigma} \mathbin{+\!\!+} [b] \end{aligned}$$

For the lock's crash-resource we only need that the locations have been persisted in some state and then the points-to predicates ending in that state.

$$\begin{aligned} P_{c,lock} \triangleq \langle \mathrm{PC} \rangle\, \exists \vec{\sigma}_x, \vec{\sigma}_y, b_y, b_x.\, & \\ & x \gtrsim_{\mathsf{p}} b_x * x \hookrightarrow_{\mathsf{na}} \vec{\sigma} \mathbin{+\!\!+} [b_x] * \\ & y \gtrsim_{\mathsf{p}} b_y * y \hookrightarrow_{\mathsf{na}} \vec{\sigma} \mathbin{+\!\!+} [b_y] \end{aligned}$$

For the location z we use the abstract state of booleans and the invariant:

$$\begin{aligned} \phi_z(\texttt{false}, v) &\triangleq v = \texttt{false} \\ \phi_z(\texttt{true}, v) &\triangleq v = \texttt{true} * x \gtrsim_{\mathsf{f}} \texttt{true} * y \gtrsim_{\mathsf{f}} \texttt{true} \end{aligned}$$

The key aspect here is that when z has the value **true** then the invariant contains the fact that x and y has been flushed in the state **true**.

In addition to the crash resource for the lock we also need a crash condition that ensures that z is available after a crash:

$$P_c \triangleq \langle \mathrm{PC} \rangle\, \exists \vec{\sigma}, b.\, z \gtrsim_{\mathsf{p}} b * z \hookrightarrow_{\mathsf{na}} \vec{\sigma} \mathbin{+\!\!+} [b]$$

The entire crash condition for the two threads is then: $P_{c,lock} * P_c$. When the lock is acquired threads are required to maintain $P_{c,lock}$ and the P_c part we let the right thread maintain.

With this setup in place the proof outline appears in Fig. 18. Note that to keep the outline simple we do not repeat resources that are unchanged in between lines in the program.

REFERENCES

Eleni Vafeiadi Bila, Brijesh Dongol, Ori Lahav, Azalea Raad, and John Wickerson. 2022. View-Based Owicki-Gries Reasoning for Persistent x86-TSO. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer, 234–261. https://doi.org/10.1007/978-3-030-99336-8_9

Lars Birkedal and Aleš Bizjak. 2020. *Lecture Notes on Iris: Higher-Order Concurrent Separation Logic.* https://iris-project.org/tutorial-material.html

Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. 2020. Understanding and optimizing persistent memory allocation. In *ISMM '20: 2020 ACM SIGPLAN International Symposium on Memory Management, ISMM 2020, virtual [London, UK], June 16, 2020*, Chen Ding and Martin Maas (Eds.). ACM, 60–73. https://doi.org/10.1145/3381898.3397212

Wentao Cai, Haosen Wen, Vladimir Maksimovski, Mingzhe Du, Rafaello Sanna, Shreif Abdallah, and Michael L. Scott. 2021. Fast Nonblocking Persistence for Concurrent Data Structures. In *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference) (LIPIcs, Vol. 209)*, Seth Gilbert (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:20. https://doi.org/10.4230/LIPIcs.DISC.2021.14

Tej Chajed. 2022. *Verifying a concurrent, crash-safe file system with sequential reasoning.* Ph. D. Dissertation. Machetutes Institute of Technology.

Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 243–258. https://doi.org/10.1145/3341301.3359632

Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 423–439. https://www.usenix.org/conference/osdi21/presentation/chajed

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2016. Using Crash Hoare Logic for Certifying the FSCQ File System. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, Ajay Gulati and Hakim Weatherspoon (Eds.). USENIX Association. https://www.usenix.org/conference/atc16/technical-sessions/presentation/chen_haogang

Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 1077–1091. https://doi.org/10.1145/3373376.3378515 ASPLOS 2020 was canceled because of COVID-19..

Kyeongmin Cho, Sung Hwan Lee, Azalea Raad, and Jeehoon Kang. 2021. Revamping hardware persistency models: view-based and axiomatic persistency models for Intel-x86 and Armv8. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 16–31. https://doi.org/10.1145/3453483.3454027

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *Proc. ACM Program. Lang.* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: strong and compositional library specifications in relaxed memory separation logic. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 792–808. https://doi.org/10.1145/3519939.3523451

Marko Doko and Viktor Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 413–430. https://doi.org/10.1007/978-3-662-49122-5_20

Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: in NVRAM data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 377–392. https://doi.org/10.1145/3385412.3386031

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

Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. 2020. go-pmem: Native Support for Programming Persistent Memory in Go. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17,* *2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 859–872. https://www.usenix.org/conference/atc20/presentation/george

Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9888)*, Cyril Gavoille and David Ilcinkas (Eds.). Springer, 313–327. https://doi.org/10.1007/978-3-662-53426-7_23

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory, See [Merchant and Weatherspoon 2019], 191–205. https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet

Artem Khyzha and Ori Lahav. 2021. Taming x86-TSO persistency. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434328

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. https://doi.org/10.1145/3009837.3009855

Arif Merchant and Hakim Weatherspoon (Eds.). 2019. *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*. USENIX Association. https://www.usenix.org/conference/fast19

Glen Mével and Jacques-Henri Jourdan. 2021. Formal verification of a concurrent bounded queue in a weak memory model. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–29. https://doi.org/10.1145/3473571

Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 265–276. https://doi.org/10.1109/ISCA.2014.6853222

Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2020a. Persistent Owicki-Gries reasoning: a program logic for reasoning about persistent programs on Intel-x86. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 151:1–151:28. https://doi.org/10.1145/3428219

Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020b. Persistency semantics of the Intel-x86 architecture. *Proc. ACM Program. Lang.* 4, POPL (2020), 11:1–11:31. https://doi.org/10.1145/3371079

Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. Weak persistency semantics from the ground up: formalising the persistency semantics of ARMv8 and transactional models. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 135:1–135:27. https://doi.org/10.1145/3360561

Pedro Ramalhete, Andreia Correia, and Pascal Felber. 2021. Efficient algorithms for persistent transactional memory. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, Jaejin Lee and Erez Petrank (Eds.). ACM, 1–15. https://doi.org/10.1145/3437801.3441586

David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. 2015. nvm malloc: Memory Allocation for NVRAM. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015, Kohala Coast, Hawaii, USA, August 31, 2015*, Rajesh Bordawekar, Tirthankar Lahiri, Bugra Gedik, and Christian A. Lang (Eds.). 61–72. http://www.adms-conf.org/2015/adms15_schwalb.pdf

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 691–707. https://doi.org/10.1145/2660193.2660243

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: a program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 867–884. https://doi.org/10.1145/2509136.2509532

Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, Rajiv Gupta and Todd C. Mowry (Eds.). ACM, 91–104. https://doi.org/10.1145/1950365.1950379

Received 2023-04-14; accepted 2023-08-27