Scaling Concurrency Reasoning to Relaxed Memory Models and Beyond

HOANG-HAI DANG, MPI-SWS, Germany

Reasoning about concurrency is hard. Reasoning about concurrency in a full-blown, non-toy language like C/C++ or Rust, which encompasses many interweaving complex features, is even harder. Yet, *realistic* concurrency involves relaxed memory models, which are significantly harder to reason about than the simple, traditional concurrency model that is sequential consistency. In order to perform verifications with realistic concurrency in such complex languages, we need a few ingredients: (1) modular reasoning so that we can compose smaller verification results into larger ones; (2) strong but abstract reasoning principles so that we can reason about tricky language features without having to deal with the tedious details of the underlying concurrency model; (3) reasoning extensibility so that we can derive new reasoning principles for both complex language features and algorithms without rebuilding our logic from scratch; and (4) machine-checked proofs with strong automation support so that we do miss potential unsoundness in our verifications. Only recently was it possible to acquire these ingredients at once, with the help of the concurrent separation logics framework Iris. In this proposal, I will present what we have been achieved in the direction of verifications for relaxed memory models. In particular, I will summarize our efforts in verifying the type system of Rust with relaxed memory models. I will then propose several further research directions as potential targets for completing this thesis. These include: (i) linearizability as stronger specifications for relaxed memory libraries; (ii) program logics for non-volatile memory models, which extend relaxed memory models; and (iii) program logics for the most relaxed architectures (e.g., ARM), i.e., the promising semantic.

1 INTRODUCTION

Reasoning about concurrency is hard, due to the explosion of possible interactions between threads running in parallel. In the traditional concurrency model of *sequential consistency* [Lamport 1979], every thread is taking turns to execute its atomic instructions, and the behavior of a concurrent program is defined as all interleavings of all threads' atomic instructions. As such, if one needs to verify some property of the program, one would need to check that property for every possible interleaving of its threads' instructions. In *coarse-grained concurrency*, where threads take turns using locks, the number of interleavings are smaller because we only need to consider interleavings of critical sections. However, in *fine-grained concurrency*, sophisticated algorithms are designed to allow multiple threads to access different parts of a data structure at the same time (using more delicate locking schemes or non-blocking operations like compare-and-swap). Therefore, the number of interleavings increases significantly. Furthermore, working with interleavings is also *non-modular*: if we want to compose our *verified* algorithms, then we would have to look at interleavings of their composition, and it is likely that our proofs for our algorithms would not be reuse-able in the proof for their composition. To reason about fine-grained concurrency, we need more *abstract* and *modular* reasoning principles.

Concurrent separation logics (hereafter, CSLs) [Brookes 2007; O'Hearn 2007] provided a feasible approach to modular control of *interferences*, such that instead of thinking with interleavings, we can reason about each thread separately and only need to abstractly take care of interferences created by other threads. This led to a series of highly expressive logics [da Rocha Pinto et al. 2014; Dinsdale-Young et al. 2010; Feng 2009; Feng et al. 2007; Fu et al. 2010; Jensen and Birkedal 2012; Jung et al. 2018b, 2015; Nanevski et al. 2014; Svendsen and Birkedal 2014; Turon et al. 2013; Vafeiadis

Author's address: Hoang-Hai Dang, MPI-SWS, Saarbrücken, Germany.

2020. 2475-1421/2020/3-ART1 $15.00
https://doi.org/

and Parkinson 2007] that have been applied to various sophisticated verification problems. Among these problems includes reasoning about relaxed memory models [Doko and Vafeiadis 2016, 2017; Kaiser et al. 2017; Turon et al. 2014; Vafeiadis and Narayan 2013] and proving soundness of the Rust's type system [Jung et al. 2018a], which are our main interests in this proposal.

Reasoning about relaxed memory models. Sequential consistency [Lamport 1979]—an interleaving semantics in which threads take turns accessing the global state, and all threads share the same view of that state—has long been the standard memory model assumed by research on concurrency verification. However, this assumption does not match the reality of modern multicore programming languages. In reality, C/C++11 (hereafter, C11) provides a relaxed memory model (hereafter, RMM) that supports a variety of different consistency levels for shared-memory accesses [Batty et al. 2011]. For programmers who demand strong synchronization, SC accesses are available, but this strength comes at the cost of disabling standard compiler optimizations and inserting expensive memory fences into the compiled code. The weaker consistency levels of *release/acquire* and *relaxed* allow one to trade off synchronization strength in return for more efficient compiled code. For example, Rust employs a variety of these different consistency levels in several of its widely-used concurrency libraries, such as `Arc`, `Mutex`, and `RwLock`.

Compared to SC, reasoning about relaxed memory is significantly more complicated: relaxed-memory programs have many more behaviors depending on which consistency levels are employed. In fact, some useful reasoning principles in SC logics are no longer sound for reasoning about relaxed behaviors. Furthermore, such behaviors are defined in C11 not in the familiar style of interleavings, but by an *axiomatic* semantics, in which the allowed behaviors of a program are defined by enumerating candidate executions (represented as "event graphs") and then restricting attention to the executions that obey various coherence axioms. Vafeiadis *et al.* overcome these challenges and provided the first abstract and modular reasoning principles for C11 in the form of various relaxed memory separation logics [Doko and Vafeiadis 2016, 2017; Vafeiadis and Narayan 2013].

However, in building these logics, Vafeiadis *et al.* were not able to use the standard model of Hoare-style program specifications from prior CSLs because notions like "the machine states before and after executing a command C" do not have a clear meaning in C11's axiomatic semantics. Instead, they had to come up with new, non-standard models of separation logic in terms of predicates on event graphs. Unfortunately, the complexity of these new models has made them challenging to adapt and extend to more complex settings, for example in verifying Rust's type system. Furthermore, although the soundness of these logics has been verified formally in Coq, there has thus far been no tool support to prove RMM programs correct in these logics.

Verifying Rust's type system. Rust [Klabnik and Nichols 2018] is a young and evolving programming language that aims to bring safety to systems programming. Specifically, Rust provides low-level control over data layout and resource management à la modern C++, while at the same time offering strong high-level guarantees (such as type and memory safety) that are traditionally associated with safe languages like Java. In fact, Rust takes a step further, statically preventing more forms of anomalous behavior, such as data races and iterator invalidation, that safe languages typically fail to rule out. Rust strikes its delicate balance between safety and control using a *substructural* type system, in which types not only classify data but also represent *ownership* of resources, such as the right to read, write, or reclaim a piece of memory. By tracking ownership in the types, Rust is able to prohibit dangerous combinations of mutation and aliasing, a well-known source of programming pitfalls and security vulnerabilities in C/C++ and Java.

Nevertheless, Rust's ownership-based type system is not always expressive enough to type-check very delicate programming idioms, *e.g.*, some pointer-based data structures, synchronization

abstractions, garbage collection mechanisms. To allow for these mechanisms, Rust supports extension to the type system via *libraries* whose implementations internally utilize *unsafe features* (*e.g.*, unchecked type casts, array accesses without bounds checks, or accesses of "raw" pointers who are untracked by the type system). Given that these libraries are not checked by the type system, it is now the responsibility of libraries developers to make sure that these extensions are actually *safe*, in the sense that they have properly encapsulated the uses of unsafe features within their "safe APIs". Unfortunately, as the language is evolving and libraries are being updated or created, it is not clear what such encapsulation formally means.

RustBelt [Jung et al. 2018a] is the first work on the formal foundations of the Rust programming language, in which it covers not only the soundness of the ownership-based type system, but also the safe encapsulation by Rust's extensions via libraries. RustBelt managed to formalize such interactions between the type system and the extensions in the presence of complex language features like recursive types and higher-order states. Furthermore, all proofs were machine-checked in Coq. Unfortunately, while ground-breaking, RustBelt assumes the sequential consistency memory model. Therefore, even though RustBelt's results increase the confidence in the safety of Rust's type system and libraries, the results cannot yet be applied to actual Rust code, which relies on the C11 memory model. In fact, in adapting RustBelt to relaxed memory, we have uncovered a data-race bug in the `Arc` library [Dang et al. 2019a] which the original RustBelt work did not find.

Challenges. As we have seen, in order to achieve more realistic guarantees for actual concurrent code, it is important to scale the reasoning principles of RMM logics to full-blown, non-toy languages like C/C++ or Rust, which encompass many interweaving complex features. To reach such a goal, we need a few ingredients: (1) modular reasoning so that we can compose smaller verification results into larger ones; (2) strong but abstract reasoning principles so that we can reason about tricky language features without having to deal with the tedious details of the underlying concurrency model; (3) reasoning extensibility so that we can derive new reasoning principles for both complex language features and algorithms without rebuilding our logic from scratch; and (4) machine-checked proofs with strong automation support so that we do miss potential unsoundness in both our logics and programs verification. Only recently was it possible to acquire these ingredients at once with the concurrent separation logics framework Iris [Jung et al. 2018b], who has strong tactics support for performing programs verification in separation logics [Krebbers et al. 2018, 2017]. Using Iris, RustBelt [Jung et al. 2018a] demonstrated that one can have modular, abstract, extensible, and machine-checked reasoning for a complex language such as Rust. Meanwhile, Kaiser et al. [2017] showed that, with Iris, it is possible to also build modular, abstract, extensible, and machine-checked reasoning for RMM.

Extending both of these works, we show that it is possible to scale modular reasoning to languages as complex as Rust even in the context of relaxed memory, and the results have been reported in the **RustBelt Relaxed** work [Dang et al. 2019a]. This requires serious changes to the models of both RustBelt and RMM logics, and I will discuss them briefly in §2. But, thanking to the modular nature of CSLs, many parts of those verifications can be reused without change.

The remaining sections will present several challenges that have not been addressed in this research direction and that are potential targets for completing this thesis: §3 discusses how *logical atomicity* [da Rocha Pinto et al. 2014; Jacobs and Piessens 2011; Jung et al. 2015] can be useful in proving stronger specifications for RMM data structures, such that we can build more complex RMM data structures from simpler ones; §4 discusses the possibility of extending RMM logics to *persistency logics* in order to support reasoning about the new technology of *non-volatile memory* [Raad et al. 2020]; and §5 discusses the necessity of supporting the remaining tricky feature

in RMM, that is *promises* [Boehm and Demsky 2014; Kang et al. 2017]. In §6, I suggest a timeline to complete the thesis with one of these directions.

2 RELAXED MEMORY FOR RUST

In [Dang et al. 2019a], we present **RustBelt Relaxed** (or $\mathbf{RB_{rlx}}$, for short), the first formal validation of the soundness of Rust under relaxed memory. Although based closely on the original RustBelt [Jung et al. 2018a], as well as iGPS [Kaiser et al. 2017] and FSL [Doko and Vafeiadis 2016, 2017] logics, $\mathbf{RB_{rlx}}$ takes a significant step forward by accounting for the safety of the more relaxed-memory operations and stricter resource reclamation schemes that real concurrent Rust libraries actually use. For the most part, we were able to verify Rust's uses of relaxed-memory operations as is. Only in the implementation of the `Arc` library did we need to strengthen the consistency level of two memory reads (from relaxed to acquire) in order to make our verification go through. And in one of these cases, our attempt to verify the original (more relaxed) access led us to expose it as the source of a previously undetected data race in the library. Our fix for this race has since been merged into the Rust codebase [Jourdan 2018].

2.1 Background on RustBelt

The initial work on RustBelt by Jung et al. [2018a] made two main contributions. First, Jung et al. proposed a formal definition of a core typed calculus called λ_{Rust}, which encapsulates the central features of the Rust language. Second, they used the Coq proof assistant to verify formally that Rust's aforementioned safety guarantees do in fact hold, both for the core λ_{Rust} calculus and for a number of widely-used Rust libraries.

In order to account for Rust's "extensible" notion of safety via unsafe language features in libraries, RustBelt employs a *semantic soundness* proof [Ahmed et al. 2010]. First, it defines a *semantic model* of Rust types: a mapping from types `T` to logical predicates on terms $\Phi(e)$, which asserts what it means for the term e to *behave* safely at type `T` (even if internally e uses unsafe features). Then, the RustBelt proof breaks into two main parts:

(1) *Safety of libraries that use unsafe features:* For any library that makes use of unsafe features, the implementation of the library is proven to satisfy the semantic model of its API, thus establishing that it is safe for clients to make use of the library. RustBelt proved safety for a number of widely-used Rust libraries, including `Arc`, `Rc`, `Cell`, `RefCell`, `Mutex`, and `RwLock`.
(2) *Safety of the λ_{Rust} type system:* The syntactic typing rules of λ_{Rust} are proven to respect the semantic model, thus establishing that code written in the "safe" fragment of Rust is in fact observably safe—*i.e.*, its behavior is well-defined.

Put together, these imply that if a program P is well-typed, and its only uses of unsafe features appear within the libraries that have been verified safe (in part 1), then P is observably safe.

RustBelt was formalized in the higher-order concurrent separation logic framework *Iris* [Jung et al. 2018b], as separation logic is a good fit for modeling Rust because it is designed around the same notion of *ownership* as Rust's type system, and thus provides built-in support for ownership-based reasoning. Iris was also designed to support the derivation of new separation logics with domain-specific reasoning principles. Jung *et al.* exploited this facility to derive a new logic called the *lifetime logic*, which they used extensively in their proofs in order to reason about Rust's "lifetimes" and "borrowing" mechanisms at a higher level of abstraction [Klabnik and Nichols 2018, §4.2, §10.3]. Furthermore, Iris's strong tactical support for developing machine-checked separation logics proofs [Krebbers et al. 2018, 2017]; this support made it possible for RustBelt to be fully mechanized in Coq.

2.2 Adaptation of RustBelt to Relaxed Memory

The overarching challenge in developing **RB$_{\texttt{rlx}}$** is that the logical foundation on which the original RustBelt is built is unsound for relaxed memory. The reason is as follows. The Iris framework (on which RustBelt is built) is parameterized by an operational semantics for the language under consideration, and depending on how this parameter is instantiated, Iris can be used to derive proof rules of varying strength. In the case of RustBelt, Iris was instantiated with a sequentially consistent (SC) semantics for λ_{Rust}. This SC instantiation of Iris (call it "Iris-SC") provides a variety of proof rules that are valid only under SC semantics and not under relaxed-memory semantics. In particular, Iris-SC enables one to establish general *invariants* governing arbitrary regions of shared memory. Unfortunately, under relaxed memory, different threads can observe writes to different locations in different orders, so one cannot in general maintain an invariant on multiple locations simultaneously.

To adapt RustBelt to relaxed memory, we must therefore rebuild it using a logic that is suitably restricted so as to be sound under relaxed memory. We followed the approach by Kaiser et al. [2017], who showed how Iris could be used to derive a relaxed-memory separation logic called iGPS, targeting RA+NA (the fragment of C11 comprising release/acquire and non-atomic accesses). iGPS, inspired by previous relaxed-memory separation logics [Turon et al. 2014; Vafeiadis and Narayan 2013], accounts for weak memory consistency by weakening the power of invariants: the user of iGPS may only establish *single-location invariants* (*i.e.*, invariants that govern a single shared memory location), the soundness of which is guaranteed by the *coherence* (or "SC per location") property of C11.

Drawing inspiration from FSL [Doko and Vafeiadis 2016, 2017], in **RB$_{\texttt{rlx}}$**, we extended iGPS further to account for the additional features of the C11 memory model that Rust libraries make use of—specifically, *relaxed accesses* and *release/acquire fences*. This extended logic is called iRC11. We then ported RustBelt so that it is built on top of iRC11 rather than Iris-SC. Following the structure of RustBelt, this porting effort breaks down into two major tasks:

Task 1: Re-prove the safety of the Rust libraries considered by RustBelt, this time verifying their real, relaxed-memory implementations in iRC11.

Task 2: Re-prove the safety of the λ_{Rust} type system, this time relying only on proof rules that are sound in iRC11.

Key challenge. As it turns out, both of these tasks require us to overcome a technical challenge that is relevant not just to Rust but to relaxed-memory verification in general: namely, that **existing work on separation logic does not provide an adequate foundation for reasoning about resource reclamation under relaxed memory**. We will first explain this challenge in the context of Task 1, before briefly describing how it also informs Task 2.

Task 1: Re-prove the safety of Rust libraries under relaxed memory. One of the main motivations for using a "systems programming" language like Rust or C/C++ (as opposed to a garbage-collected language like Java) is to have more precise control over limited resources such as memory. In particular, the Rust programmer can be assured that when an object goes out of scope, the destructor (`drop` method) associated with its type will be invoked and any resources it owns will be reclaimed. Yet the safety of destructors is often quite subtle because objects can contain references to resources that are shared with other objects. For example, objects of type `Arc<T>` are simply aliases to a shared **`struct`** containing an object of type `T` along with a *reference counter*, which keeps track of the current number of active aliases to the object. Consequently, the destructor for `Arc<T>` cannot simply reclaim the shared **`struct`** that it points to: rather, it decrements the shared reference

$$\text{SC-CInv-Acc}\ \frac{\{I * P\}\, e\, \{v.\, I * Q\} \qquad \text{phys\_atomic}(e)}{{}^{\tau}\boxed{I} \vdash \{[\tau]_q * P\}\, e\, \{v.\, [\tau]_q * Q\}} \qquad \text{SC-CInv-Tok}\ [\tau]_{q+q'} \Leftrightarrow [\tau]_q * [\tau]_{q'} \qquad \text{SC-CInv-Cancel}\ {}^{\tau}\boxed{I} \vdash [\tau]_1 \Rrightarrow I$$

Fig. 1. Key rules for cancellable invariants in Iris-SC.

counter, and only if it observes that it was the last remaining alias can it safely reclaim the memory for the reference counter and invoke the destructor for the object of type `T`.

RustBelt showed how to put this subtle kind of resource reclamation on a sound formal footing using Iris-SC's mechanism of *cancellable invariants* (Figure 1), a generalization of Gotsman et al. [2007] and Hobor et al. [2008]'s "storable locks". A cancellable invariant ${}^{\tau}\boxed{I}$ is an invariant governing a shared resource (described by proposition I) which is only "active" for a certain period of time, after which point it is "cancelled". To access the shared resource during an atomic step of computation (SC-CInv-Acc), a thread must prove that the invariant is still active by exhibiting ownership of an *invariant token* $[\tau]_q$, where q is a fraction in (0,1]. This is an instance of the well-known concept of *fractional permissions* [Boyland 2003], and correspondingly, ownership of invariant tokens can be split or combined through fractional arithmetic (SC-CInv-Tok). If a thread π can assert ownership of $[\tau]_1$ (*i.e.*, the "full" τ token), it knows that no other thread can assert that the invariant is active; thus it is safe for π to cancel the invariant and reclaim full ownership of I (SC-CInv-Cancel), after which it can free the memory governed by I if it wants to. In RustBelt, cancellable invariants played a crucial role in verifying the safety of destructors such as `Arc`'s.

However, adapting cancellable invariants to the relaxed-memory setting turns out to be quite tricky—tricky enough that no existing relaxed-memory separation logic supports them.[1] Even if, following iGPS and its predecessors, we restrict invariants to govern a single location, a problem arises in how to model the cancellable invariant *tokens*. Under SC, one can simply model invariant tokens as a form of *ghost state*, *i.e.*, purely logical state that is manipulated by the proof but does not appear in the physical program. But in existing relaxed-memory separation logics, ghost state is *unsynchronized*, meaning that ownership of it can be transferred between threads without the need for any physical synchronization. On the one hand, unsynchronized ghost state is indispensable for representing *globally consistent* state, such as (in the case of `Arc`) the number of `Arc` aliases currently in existence. On the other hand, if invariant tokens are modeled naively as unsynchronized ghost state, the logic of cancellable invariants becomes unsound!

Our solution is to instead model invariant tokens using a novel notion of *synchronized ghost state*: ghost state that implicitly tracks the subjective view of the thread that owns it, and that therefore can only be transferred between threads using physical synchronization. **Using synchronized ghost state, iRC11 offers the first general account of resource reclamation in relaxed-memory separation logic. We have demonstrated its effectiveness on a number of real Rust libraries.**

Task 2: Re-prove the safety of the λ_{Rust} type system under relaxed memory. In contrast to RustBelt's proofs of safety for libraries, its proof of safety for the λ_{Rust} type system did not rely directly on cancellable invariants or any other SC-specific features of Iris-SC. Rather, as mentioned above, the safety proof for the type system made essential use of a Rust-oriented logic called the *lifetime logic*, which was a domain-specific logic derived within Iris-SC. Thus, if we are able to show that the

[1] iGPS supports a related notion of "fractional protocol", but it is not nearly as powerful as cancellable invariants and is thus not general enough to account for resource reclamation in Rust.

lifetime logic remains sound under relaxed memory—by instead deriving its soundness in iRC11—then **RB$_{\texttt{rlx}}$** can inherit RustBelt's safety proof for the λ_{Rust} type system without modification!

Synchronized ghost state is the key to making this modular porting strategy possible. Specifically, the lifetime logic is centered around a mechanism called *borrow propositions*, describing resources that are borrowed for the duration of a Rust "lifetime" and that can be reclaimed once the lifetime is over. Borrow propositions are similar in many ways to cancellable invariants, but also more flexible and more complex in terms of the protocols they support for sharing and reclamation of resources. Just as synchronized ghost state enables us to adapt cancellable invariants to relaxed memory, it plays an analogously central role in adapting borrow propositions to relaxed memory as well.

2.3 Contributions of RustBelt Relaxed

RB$_{\texttt{rlx}}$—an adaptation of RustBelt to a relaxed memory model—is fully mechanized in Coq (like its predecessor). A summary of contributions of that work is as follows.

- We define ORC11, a new *operational-semantics-based* characterization of a large fragment of C11, including release/acquire/relaxed/non-atomic accesses and release/acquire fences.[2] Developing such an operational semantics for C11 is a necessary prerequisite for instantiating the Iris framework. Since the C11 model is known to be flawed [Boehm and Demsky 2014], we instead design ORC11 to match the semantics of RC11 (Repaired C11) [Lahav et al. 2017], and in the appendix [Dang et al. 2019b] we sketch a proof of correspondence between them.
- We develop iRC11, a logic for ORC11 derived within Iris, which combines elements of iGPS and FSL, and moreover supports resource reclamation via cancellable invariants in a manner that is sound for relaxed memory. The soundness of iRC11 relies crucially on our novel construction of *synchronized ghost state*.
- We use iRC11 to port RustBelt from SC to relaxed memory. In particular, the major components that required re-verification were the library proofs (since we are now verifying implementations with relaxed-memory operations in them) and the proof of soundness of RustBelt's lifetime logic. The proof of safety of λ_{Rust}'s type system, by virtue of being built atop the lifetime logic, did not need to be changed at all.

The adaptation involves many components whose full technical explanation is beyond the scope of this proposal. I refer the reader to the main paper [Dang et al. 2019a] for more details.

3 STRONG SPECIFICATIONS FOR RMM DATA STRUCTURES

One common question with reasoning about concurrent libraries is that "What is the strongest specification we can prove for a library?" This is particularly important when our library is being used by a client to build a new library, where the client would rely on certain strong properties of *our* library to prove the specification for *their* library.

For example, consider the following specification of stacks.

$$\{\mathsf{isStack}(s, P) * P(v)\}\ \mathsf{push}(s, v)\ \{\mathsf{isStack}(s, P)\} \qquad \text{(STACK-PUSH)}$$

$$\{\mathsf{isStack}(s, P)\}\ \mathsf{pop}(s)\ \{v.\ \mathsf{isStack}(s, P) * \mathbf{if}\ v \neq \mathsf{EMPTY}\ \mathbf{then}\ P(v)\ \mathbf{else}\ \mathsf{emp}\} \qquad \text{(STACK-POP)}$$

This specification ties a stack s to a predicate $P : \mathit{Val} \to \mathsf{Prop}$ from values to assertions, which defines the resource $P(v)$ that will be transferred from a call of $\mathsf{push}(s, v)$ to its matching successful $\mathsf{pop}(s)$. This *per-element* specification captures the synchronization between a push and its matching successful pop, which guarantees the soundness of transferring the resource $P(v)$. However, the specification does not capture any property that relates the stack's operations other than matching

[2] Caveat: ORC11 omits SC accesses because (1) they are not used by any of the libraries verified in RustBelt, and (2) it is still an open question how to develop a separation logic for reasoning about SC accesses in a relaxed-memory setting.

pairs of push and pop. In particular, it cannot guarantee a "historical" property, in which what one has observed on the library restricts what one can do with the library afterwards. Such a property can be seen in the following Message-Passing example [Raad et al. 2019a] using a stack and a queue.

$$\begin{array}{l|l} \mathsf{enqueue}(q,1); & \textbf{if}\ \mathsf{pop}(s)=1\ \textbf{then} \\ \mathsf{push}(s,1) & \quad a := \mathsf{dequeue}(q) \qquad a \neq \mathsf{EMPTY} \end{array} \tag{MP-LIB}$$

In MP-LIB, we are using the stack s to send a message from the left thread to the right thread that the enqueue of 1 into the queue q has happened. Therefore, it should be the case that, assuming there is no other dequeue, the dequeue of q by the right thread cannot return EMPTY. The "historical" property here is the fact that the right thread has observed the enqueue and such event restricts what the thread can do with the queue afterwards. Unfortunately, this property cannot be verified with only STACK-PUSH/STACK-POP and the similar per-element specification of the queue.

$$\{\mathsf{isQueue}(q,P) * P(v)\}\ \mathsf{enqueue}(q,v)\ \{\mathsf{isQueue}(q,P)\} \tag{QUEUE-ENQ}$$

$$\{\mathsf{isQueue}(q,P)\}\ \mathsf{dequeue}(q)\ \{v.\ \mathsf{isQueue}(q,P) * \textbf{if}\ v \neq \mathsf{EMPTY}\ \textbf{then}\ P(v)\ \textbf{else}\ \mathsf{emp}\} \tag{QUEUE-DEQ}$$

The cause of the problem is clear from QUEUE-DEQ: the dequeue specification only considers that q is in fact a queue ($\mathsf{isQueue}(q,P)$), but it does not consider the relation of the dequeue with the history of q—which may contain enqueues that the caller has previously observed.

Furthermore, the similarity between the per-element specifications of the stack and the queue also makes another problem apparent: they cannot distinguish between a stack and a queue! This is because other than capturing the synchronization between matching push-pop's and enqueue-dequeue's, the specifications cannot express that the stack's history follows the LIFO property and the queue's history follows the FIFO property. Unfortunately, most recent RMM logics do not have support to derive such a stronger specification.

Linearizability. In the SC context, specifying properties over the whole history of a library's operations has been achieved through linearizability [Herlihy and Wing 1990]. Intuitively, a concurrent library is *linearizable* if every execution of its operations can be abstracted into (read: *simulated by*) a *sequential execution* or *history*. Then the behaviors of the current library can be specified easily as properties (for example, LIFO) over its sequential histories. Unfortunately, the requirement of sequential execution dictates the existence of a *total order* over the history which does not always exist for RMM libraries [Hemed et al. 2015; Raad et al. 2019a]. As such, linearizability is only applicable to strongly consistent RMM libraries, *e.g.*, those whose every operation is synchronizing with every other operation.

For RMM libraries that do not have a total order on histories, one can only hope to specify enough restrictions on the histories of operations through *partial orders*. Raad et al. [2019a] are the first to propose a formal framework—but not a high-level program logic—to specify properties over such partial orders for RMM libraries, following the axiomatic-style memory model of C11. In this section I propose to combine this style of specification (§3.1) with *logical atomicity* (§3.2)—an approach to encode linearizability in concurrent separation logics—in order to construct a high-level, abstract concurrent separation logic that supports stronger specifications for RMM libraries.

3.1 RMM Specifications with Histories and Partial Orders

In the relaxed memory setting, Raad et al. [2019a] are the first to propose a formal framework to encode modular library specifications as partial orders on the histories of library operations. Following the axiomatic-style C11 memory model, they specify a RMM library by (1) defining

the set of library events that can be generated by the library's operations; then (2) enumerating candidate histories of the library, represented as event graphs (similarly to C11); and (3) defining the accepted behaviors as only histories that satisfy certain library-specific axioms. For example, assume that operations of a queue generate the events $\mathsf{enq}(q, v)$—an enqueue of v, $\mathsf{deq}(q, v)$—a successful dequeue of v, and $\mathsf{deq}(q, \text{EMPTY})$—an unsuccessful dequeue. Then a reasonably strong specification of queues who requires the FIFO property would accept the history $\mathsf{enq}(q, 1); \mathsf{enq}(q, 2); \mathsf{deq}(q, 1)$,[3] but reject the history $\mathsf{enq}(q, 1); \mathsf{deq}(q, 2)$. The library-specific axioms also allow for the flexibility in linearizability: if a library is linearizable, then its specification can include a total-order axiom; otherwise, the library can only specify enough restrictions over its histories with the partial orders. For example, in their *Weak Queue* specification, Raad et al. [2019a] only require that ordered enqueues must be dequeued in the same order, allowing the possibility that some enqueues can be *unordered* and thus one cannot say much about their matching dequeues.

To express such axioms, Raad et al. [2019a]'s specifications include, among other relations, a *synchronization order* so. As the name suggest, this partial order should be used by the specifier to denote pairs of library events (representing operations) that have physical synchronizations. Similarly to C11, the synchronization order together with the *program order* po contributes to the *happens-before order* hb, which ultimately decides what events should be ordered with one another. Then, linearizability of a library can be expressed as there exists a strict total order to that agrees with hb: $\mathsf{hb} \subseteq \mathsf{to}$. In the case of a *non-linearizable* queue, we can instead specify that pairs of enqueue events $\mathsf{enq}(q, v)$ and their matching successful dequeue events $\mathsf{deq}(q, v)$ should be included in the queue's so. We can then encode the condition that "ordered enqueues cannot be dequeued out of order" as: if $(\mathsf{enq}_1, \mathsf{deq}_1), (\mathsf{enq}_2, \mathsf{deq}_2) \in \mathsf{so}$ *and* $(\mathsf{enq}_1, \mathsf{enq}_2) \in \mathsf{hb}$ then $(\mathsf{deq}_2, \mathsf{deq}_1) \notin \mathsf{hb}$. Obviously, such a condition is not captured in a per-element specification like QUEUE-ENQ and QUEUE-DEQ.

Raad et al. [2019a]'s framework is also general enough to allow a client to compose multiple library specifications and to derive that, due to sufficient synchronizations, certain behaviors of the client cannot happen, as in the MP-LIB example.

3.2 Strong RMM Specifications with Logical Atomicity

While general, Raad et al. [2019a]'s framework has a problem with scalability: its reasoning is too low level and thus would require too much formalization effort. In particular, their verifications, albeit done in Coq, work at the level of library events and their relations. With the axiomatic style of C11, one has to work with the complete execution at once, and tries to construct relations over the execution such that they satisfy the library-specific axioms. That is, even though the specifications are modular, their verifications are still somewhat global because we still need to reason about complete *traces*. In solving this problem, I propose the following research questions:

- Can we lift Raad et al. [2019a]'s library specifications to a program logic, where we can exploit powerful features of Hoare-style concurrent separation logics to reason about histories more *incrementally* and *abstractly*?
- Can we derive more abstract reasoning principles for library specifications through *ownership*? For example, similarly to how RMM logics have derived the abstract reasoning of *single-location* invariant, can we also derived *single-library* invariants?
- Can the more abstract reasoning principles with CSLs really simplify the verifications of implementations with respect to library specifications? For example, can we significantly reduce Raad et al. [2019a]'s 2KLOC verification of a queue implementation?

[3] Note that the specification should accept this sequential history, but it may also accept non-sequential histories.

- In fact, can we prove or at least *demonstrate* that our library specifications are much stronger than existing specifications, in the sense that those specifications can be derived from our specifications?[4]

In a way, the motivation for this problem is in the same spirit with that of Kaiser et al. [2017]'s iGPS: can we build a more abstract and extensible framework with strong mechanization support for reasoning about RMM libraries? In other words, can we lift Kaiser et al. [2017]'s approach from C11 to library level? I propose to achieve that goal by combining partial-ordered histories with logical atomicity—an approach to encode linearizability in concurrent separation logics. Following a preliminary investigation in collaboration with Mansky [2020], I anticipate the following challenges.

Challenge 1: *Proving logical atomicity for relaxed memory.* The first challenge is how to soundly abstract a concurrent operation to a single event of a history, where a library operation can be composed of multiple instructions and thus is not *physically atomic*. Fortunately, it is very commonly the case that a concurrent operation is *logically atomic* [da Rocha Pinto et al. 2014; Jacobs and Piessens 2011; Jung et al. 2015], in the sense that even the operation performs several instructions, its effect is committed atomically by a single instruction, which we call the *commit point* of the operation. That is, the effect of the operation appears atomically to other concurrent observers of the data structure. For example, imagine the implementation of the Treiber stack with linked lists: the push operation needs several instructions to create a new node and link it to the current list, but the effect of push only becomes visible to others when the head of stack is updated (physically) atomically to the new node. The update of the stack's head is push's commit point, and is implemented with an atomic compare-and-swap (CAS). Logically, we can see the commit point as the point where the operation publishes its change into the global history of the data structure. That is why we can soundly abstract the effect of an operation as a single event in the history: the event is inserted into the history at the commit point.

Logical atomicity in Iris. In the SC context, the commit point is often referred to as the *linearization point*, because the history is supposed to be linearizable. In fact, linearizability is reflected into Iris through its implementation of logical atomicity. In the logic of Iris, the main power of logically atomic operations is that they can be used with stronger proof rules that normally only applicable to physically atomic instructions. For example, recall that the *invariant access* rule SC-CInv-Acc (§2.2) allows one to access the shared resources I stored in the invariant, but only for the duration of a *physically* atomic expression e. So such a rule is not applicable to non-atomic operations. However, a similar rule is applicable to a logically atomic operation, with the access to the invariant happening *around* the operation's linearization point.[5] More specifically, Iris supports the logically atomic Hoare triples of the form $\langle P\rangle\ e\ \langle Q\rangle$, where P and Q are not standard pre- and post-conditions, but are pre- and post-conditions that e has access to *around* its linearization point. That is, e is not supposed to transform P to Q around its whole execution, but only to transform P to Q around its committing physically atomic instruction. As such, the logically atomic triples admit the following invariant access rule.

$$\textsc{LogAtom-Inv-Acc}\quad \frac{\langle I * P\rangle\ e\ \langle I * Q\rangle}{\boxed{I} \vdash \langle P\rangle\ e\ \langle Q\rangle}$$

[4] It would be a great result if we can formally prove that our specifications with histories are the *strongest*, but this appears very ambitious to me.

[5] This is a simplification. The Iris implementation of logical atomicity actually supports accessing the invariant many times before the linearization point, in order to allow for *retries*.

While this rule has proven useful in "internalizing" linearizability proofs in Iris-SC, it is obviously unsound in relaxed memory. This is due to the fact that, as discussed in §2.2, general invariants that may span over multiple locations are unsound in relaxed memory, where different threads can observe writes to different locations in different orders. We can recover the rule by restricting to single-location invariants (again, see §2.2) or *objective invariants* where the invariant content must be objective, meaning that its truthiness is independent of threads' observations. For example, *unsynchronized ghost state* is objective, and indeed can be used to encode the global history of the data structures (see also **Challenge 2** below). However, preliminary investigation shows that logically atomic specifications with objective invariants are not as strong as we would like, because they cannot tie the physical synchronization information into the purely-ghost history!

It is also not obvious how useful logically atomic specifications with single-location invariants would be for RMM data structures. In fact, I believe that logically atomic specifications as in the current form are not very useful for RMM logics. I instead suggest that such logically atomic specifications would be useful in the *base logics* of those logics. More specifically, Kaiser et al. [2017]'s approach—which we follow—to encode RMM logics in Iris consists of two steps: (1) building a base logic in Iris where thread-local observations as well as synchronization information are explicit in the form of *views*; and then (2) deriving the abstract top-level logic (for example, with iGPS single-location invariants) on top of the base logic where the tedious views are hidden. As such details are hidden in top-level logic, it makes more sense to prove specifications that capture synchronization information at the level of the base logic.

Example logically atomic specification for queues. Below we propose a logically atomic specification for queues at the base logic in Iris where views are explicit.

QUEUE-LogAtom-Deq

$$[\![\mathsf{QueueLocalView}(q, \mathcal{G}_0, G_0)]\!](V) \vdash \begin{array}{l} \langle \forall \mathcal{G} \sqsupseteq \mathcal{G}_0.\, \mathsf{History}(q, \mathcal{G}) \rangle \\ \quad \mathsf{dequeue}(q)\ @\ V \\ \left\langle (v, V').\, V' \sqsupseteq V \land \exists \mathcal{G}' \sqsupseteq \mathcal{G} \cup \{\mathsf{deq}(q, v)\}, G' \sqsupseteq G_0. \right. \\ \qquad \left. \mathsf{History}(q, \mathcal{G}') * [\![\mathsf{QueueLocalView}(q, \mathcal{G}', G')]\!](V') * \ldots \right\rangle \end{array}$$

Here, the thread-local views V and V' are explicit: V is the local view of the caller thread at the call of the function, while V' is its local view after the call. Notice that views can only grow: $V \sqsubseteq V'$. In this specification, the function requires access to the true, current history $\mathcal{G}$ of the queue through the resource $\mathsf{History}(q, \mathcal{G})$. The history $\mathcal{G}$ is an event graph that is equipped with partial orders as in Raad et al. [2019a]. $\mathsf{History}(q, \mathcal{G})$ restricts $\mathcal{G}$ to be *consistent*, which includes the properties we want to for the history, for example that matching enqueues and dequeues are synchronizing, or that the history has the FIFO property.

The specification says that it will update the history to a bigger history $\mathcal{G}'$ ($\sqsupseteq \mathcal{G}$) that would contain the new dequeue event $\mathsf{deq}(q, v)$. The specification also takes into account the caller thread's previous observations through the view-dependent assertion $\mathsf{QueueLocalView}(q, \mathcal{G}_0, G_0)$, where $\mathcal{G}_0$ is a snapshot (of the actual history $\mathcal{G} \sqsupseteq \mathcal{G}_0$) that the thread has *logically* observed, while G_0 is the subgraph of $\mathcal{G}_0$ that the thread has *physically* observed. Intuitively, as the name suggests, $\mathsf{QueueLocalView}(q, \mathcal{G}_0, G_0)$ records the thread's local view on events of the queue q, which will restrict what future behaviors the thread can observe on q.

The client of QUEUE-LogAtom-Deq, exploiting LogAtom-Inv-Acc, can then put $\mathsf{History}(q, \mathcal{G})$ inside an Iris general invariant, together with its own extra invariant $\mathsf{ClientInv}(q, \mathcal{G}, \ldots)$ that can enforce further restrictions or *protocols* on how the queue will be used, or can attach more resources

to the queue, to derive a more specific specification:

$$\boxed{\exists \mathcal{G}.\, \mathsf{History}(q, \mathcal{G}) * \mathsf{ClientInv}(q, \mathcal{G}, \ldots)} \vdash$$
$$\{[\![\mathsf{QueueLocalView}(q, \mathcal{G}_0, G_0)]\!](V)\}$$
$$\quad \mathsf{dequeue}(q) \ @ \ V$$
$$\left\{ \begin{array}{c} (v, V').\, V' \sqsupseteq V \wedge \exists \mathcal{G}' \sqsupseteq \mathcal{G}_0 \cup \{\mathsf{deq}(q, v)\}, G' \sqsupseteq G_0. \\ [\![\mathsf{QueueLocalView}(q, \mathcal{G}', G')]\!](V') * \ldots \end{array} \right\}$$

It should be possible to abstract this specification further to the top-level logic, in the same style as that of [Dang et al. 2019a; Kaiser et al. 2017], where views are hidden away. Such specification would then be not only strong enough but also easy enough to use to verify client programs that compose libraries, for example MP-LIB.

In summary, I propose to prove strong library specifications through logically atomic specifications at the level of the base logic in Iris, where the synchronization information as well as Iris general invariants are available. I will also explore to see whether the verifications of implementations with respect to specifications can still be done in the top-level logic *i.e.*, parts of the proofs can still avoid dealing with tedious base logic details and can instead take advantages of the abstract reasoning principles from the top-level logic.

Challenge 2: *Encoding histories as ghost state and library axioms as invariants.* In order to state and verify logically atomic library specifications, I will encode library histories as ghost state in Iris. Note that as we are working prefixes of program traces in Iris, we do not assume a complete history and then building partial orders on it. Here, we are expected to build the history and its partial orders *incrementally*, step-by-step (but the history does not need to grow in a single, linear list). I would need to handle several technical questions.

- I will explore the suitable ghost structures (partial commutative monoids, or CMRAs [Jung et al. 2018b]) for histories. The structure should be general enough to encode partial orders on the history, and also to support agreements between history observations, as the history must be shared between threads and thus they must have compatible observations over the history.
- I will need to decompose library-specific axioms, which are stated on complete histories, into *inductive invariants* over the growing histories, such that library consistency is maintained at every operation (when a new event added), and also ultimately at the end when we have the complete history.
- I will investigate how to tie physical synchronization information into the histories to allow for strong specifications, the like of which can be used to derive more abstract specifications (see **Challenge 3** below).

Remark. In the more general context of RMM, the commit point is simply the point where the event is inserted into the history, and how the event is ordered with other events is determined by the various partial orders that can be updated together with the insertion of the event, or can also be updated *in the future* when new events are added to the history. It is then also interesting to explore the relation between partial ordered histories and prophecy variables [Abadi and Lamport 1991; Jung et al. 2020].

Challenge 3: *Deriving high-level, abstract library specifications.* The logically atomic library specifications would be expressive enough to capture various stronger properties of a library's histories. However, as we expect them to be stated at the base logic level with explicit threads observations and synchronization information, they would be tedious to work with. The question

is then can we derive more abstract specifications for libraries from the "base-logic" specifications? This is very similar to how [Kaiser et al. 2017] derived more abstract reasoning principles for C11 in the top-level logic. At the time of this writing, it is unclear to me what such abstract reasoning principles for library-level would be. I will explore whether single-library invariants in the style of iGPS and iRC11's single-location invariants would be useful. Such argument also needs to be supported by several representative examples, which must include libraries that are constructed from smaller libraries.

4 PERSISTENCY LOGICS FOR NON-VOLATILE MEMORY

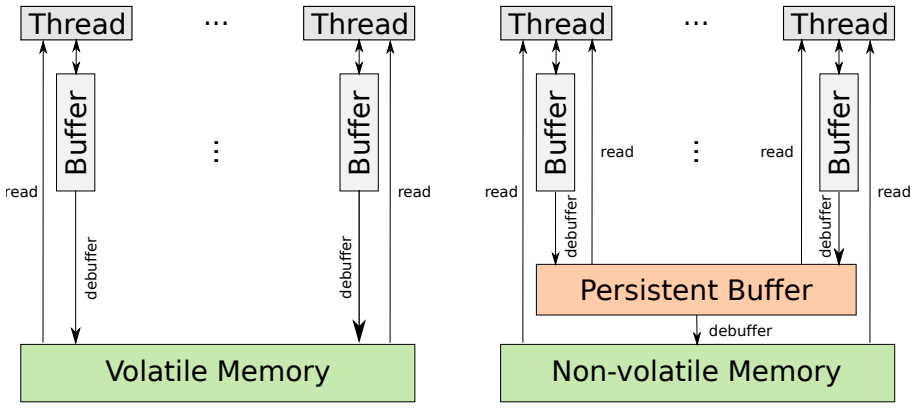
Non-volatile memory (NVM) refers to a set of emerging technologies [Boehm and Chakrabarti 2016; Gogte et al. 2018; Intel 2014; Kolli et al. 2017; Pelley et al. 2014; Raad et al. 2020, 2019b] that promise comparable performance to volatile RAM, but also guarantee memory *persistence* on disk beyond power failures. That is, after a crash, the memory can be recovered and computations can be resumed quickly. NMV—that is, persistent memory— is expected to eventually replace volatile memory for fast access to persistent data.

However, using persistent memory *correctly*, so as to maintain reasonable consistency after recovery, is not easy. In order to achieve a desirable state after recovery, one is required to understand how writes are propagated to memory. Modern multi-core architectures already provide hierarchies of volatile caches between CPUs and the volatile memory, which affects how writes are propagated one processor to other processors and the volatile memory. Persistent memory introduces further *persistent caches*, which affects how writes are persisted to the non-volatile memory. Therefore, writes may not be persisted at the same time and in the same order as when they are issued by processors, and can lead to surprising behaviors. To properly exploit persistent memory, one now has to understand not just *memory consistency*—the order of writes propagation between processors, as abstracted by relaxed memory models, but also *memory persistency*—the order of writes persistency to memory, now abstracted by memory *persistency models*.

There have been several proposals for persistency models with varying strength and performance [Gogte et al. 2018; Izraelevitz et al. 2016; Joshi et al. 2015; Kolli et al. 2017, 2016; Raad et al. 2020, 2019b]. But only until recently was the persistency semantics of the mainstream Intel x86 architecture [Intel 2019] formalized by Raad et al. [2020]. They developed the Px86 (*persistent x86*) model in both axiomatic and operational forms, by extending the relaxed memory model x86-TSO [Sewell et al. 2010] with persistency semantics. Raad et al. [2020] formalized the semantics in close collaboration with Intel's research engineers and followed the Intel reference manual [Intel 2019], allowing for the clarification of several ambiguities in the manual text. The disambiguated semantics was then formalized in the Px86_{sim} model, which is the target of this section's proposal: I propose to explore the research question to achieve *modular, abstract, and machine-checked verifications of programs for persistent programs in* $Px86_{sim}$ *with concurrent separation logics*. In §4.1, I will give a quick review of persistency semantics, and in §4.2, I will discuss the possible challenges in resolving this research question.

4.1 Persistency Semantics

As briefly mentioned above, memory consistency models define the order in which the effects of memory instructions are made visible to other threads (processors). This order is called the *consistency order*. Meanwhile, memory persistency models define the *persistency order*, *i.e.*, the order in which the effects of of memory instructions are committed to persistent memory and thus can be recovered after a crash. Naturally, the two orders do not need to coincide or agree. That is, the effects of memory instructions can appear to threads differently from how they are persisted. In



(a) x86-TSO subsystem (b) $Px86_{sim}$ subsystem

Fig. 2. Comparing x86 relaxed memory model (x86-TSO) and persistency memory model ($Px86_{sim}$).

fact, forcing them to agree would hinder performance. Allowing them to disagree, however, makes reasoning harder.

To make it more concrete, let us look at the consistency order and persistency order of the Intel-x86 architecture. An illustration of the original x86-TSO model as well as its persistent extension $Px86_{sim}$ Raad et al. [2020] is given in Figure 2. Both models use various *buffers* to allow for delaying effects to reach threads or the persistent memory.

The x86-TSO model. The Intel-x86 architecture's consistency order follows the *total store order* (TSO) [Sewell et al. 2010]. In this context, we then also use the *store order* to refer to the consistency order. In the x86-TSO model, each thread is associated with a FIFO store buffer (see Figure 2a). When a thread issues a write, the write first goes into the store buffer. At non-deterministic points in time, writes in the store buffer will be debuffered in FIFO order and propagated to the main volatile memory. When a thread issues a read of a location X, it first finds the *last* buffered write to X in its own buffer. If such a write exists, the thread reads the value from that write. Otherwise, the thread reads the value of X directly from the volatile memory. As the writes are *delayed* while the reads are executed immediately, this allows for the *Store Buffering* (SB) behavior, where it appears as if an earlier write were *reordered after* a later read:

$$\begin{array}{l|l} X := 1; & Y := 1; \\ a := !Y & b := !X \end{array} \qquad \text{(SB)}$$
$$a = 0 \wedge b = 0$$

In this example, a and b are local variables, and both (global) locations X and Y are initialized with 0. In both threads, the write is delayed in the current thread's store buffer, and then the read is executed immediately. As the read is not able to find a write in the current thread's store buffer, it consults the volatile memory and thus reads 0. More specifically, in the left thread, the buffer only contains the write of 1 to X and no writes to Y, so the thread consults the volatile memory, at which point the write to Y by the right thread is also buffered and has not reached the volatile memory, ultimately resulting in the left thread reading 0. This appears as if in both threads the write is reordered after the read.

To prevent such reordering behavior, programmers can use **`mfence`** instructions to flush the store buffers and debuffer all delayed writes to the memory. In (SB), by inserting **`mfence`** between the

$X := 1;$ $Y := 1;$ (a)	$X := 1;$ **flush** $X;$ $Y := 1;$ (b)	$X := 1;$ ‖ $a := !Y;$ **flush** $X;$ ‖ **if** $a = 1$ **then** $Y := 1;$ ‖ $Z := 1$ (c)
rec: $X, Y \in \{0, 1\}$	**rec**: $Y = 1 \implies X = 1$	**rec**: $Z = 1 \implies X = 1$

Fig. 3. Examples of Px86_{sim} programs with recovery invariants. X, Y and Z are distinct locations on different cache lines. Initially $X = Y = Z = 0$.

writes and the reads in both threads, we prevent the writes from being reordered after the reads, thus making the final behavior $a = 0 \wedge b = 0$ impossible.

The Px86_{sim} model. Raad et al. [2020] extends x86-TSO with a *persistent buffer* (see Figure 2b) to model the persistency order in Px86_{sim}. The persistent buffer contains pending writes that are to be persisted to the non-volatile memory. Note that the main memory is now non-volatile and will persist beyond crashes, while the persistent buffer is still volatile and cannot be recovered after a crash. Similar to x86-TSO, writes first go into the thread-local store buffers, then are debuffered from store buffers to the persistent buffer at non-deterministic points in time, and then debuffered from the persistent buffer to be persisted in the memory, also at non-deterministic points in time. In Intel's persistency semantics, writes on different locations may persist in any order, while writes on the same location persist in the *store order*, *i.e.*, the order in which writes arrive in the persistent buffer. Therefore Px86_{sim} models the persistent buffer as a queue that propagates writes on the *same* location in the FIFO order, but propagates writes on *different* locations in an arbitrary order.

Reads also need to follow this hierarchy of buffers. When reading from a location X, a thread first finds the *last* buffered write to X in its own store buffer. If such a write exists, the thread reads the value from that write. Otherwise, it proceeds to find the *last* buffered write to X in the persistent buffer. If such a write exists, the thread reads the value from that write. Otherwise, it reads the value of X from the non-volatile memory.

Figure 3 shows several examples that demonstrate some interesting behaviors of the Px86_{sim} semantics. Each example satisfies a *recovery invariant* **rec** which constrains the possible values of locations in the non-volatile memory at the recovery time after a crash. In Figure 3a, after a crash, the values of X and Y can be any from $\{0, 1\}$, because the persistence of distinct locations X and Y can happen in arbitrary order. For example, it is possible that $X = 0$ and $Y = 1$, because the write of 1 to Y was persisted before the crash, but the write of 1 to X was not. Note that while this behavior is allowed in the persistency order, it is disallowed in the consistency (store) order which requires a total store order: the writes cannot be observed by threads out-of-order. More specifically, if a thread sees the write of 1 to Y, it must have seen the write of 1 to X.

In Figure 3b, we can use a *persist instruction* **flush** to force the ordering of persisting between X and Y. This ensures that the write of 1 to X must be persisted before the write of 1 to Y is persisted, thus the corresponding recovery invariant. Note that **flush** does not force the persistence immediately, it only enforces ordering: persistence happens asynchronously, at some later time. So if the program crashes right after the code has finished executing, the persistence of the writes does not necessarily happen, and the recovery state can contain $X = Y = 0$. This is why the recovery invariant is conditional on the value of Y.

Figure 3c combines the consistency order through *message-passing* with Y and **flush** to force the ordering of persisting X and Z. If the right thread reads 1 from Y, x86-TSO enforces that

$X := 1$; **flush** X must happen before $Z := 1$. As such, **flush** X enforces that $X := 1$ is persisted before $Z := 1$, so if after a crash $Z = 1$ then $X = 1$. Note that there is still no relation between the persisting of Y and Z: even if $Z := 1$ is persisted, there is no guarantee that $Y := 1$ is also persisted. Therefore, we cannot have the recovery invariant ($Z = 1 \implies X = 1 \land Y = 1$).

More details on the persistency behaviors of other instructions can be found in [Raad et al. 2020].

4.2 Persistency Concurrent Separation Logics

While there have been several proposals for the formalization of persistency semantics, unfortunately there has been no framework to formally verify algorithms and libraries built on those persistency semantics. Several persistent data structures (like queues, maps, sets) [Friedman et al. 2018; Nawab et al. 2017; Raad and Vafeiadis 2018; Zuriel et al. 2019] only provide pen-and-paper correctness proofs, or assume sequential consistency as the consistency model, or additionally reason globally with traces. Through personal communication with Raad [2020], I learned that they were extending the OGRA (Owicki-Gries for Release-Acquire) proof system [Lahav and Vafeiadis 2015] in order to a build *rely-guarantee*-style Owicki-Gries proof system for the Px86_{sim} semantics. However, they were not targeting a machine-checked framework to perform Px86_{sim} program verifications.

Our goal is to build a modular, abstract, and machine-checked verification framework for persistent algorithms. To achieve this goal, we have to simultaneously explore reasoning principles for both the consistency model *and* the persistency model—which do not always agree, while asking ourselves whether concurrent separation logics are the right tool to construct such modular and abstract principles. Previous relaxed memory CSLs [Dang et al. 2019a; Doko and Vafeiadis 2016, 2017; Kaiser et al. 2017; Turon et al. 2014; Vafeiadis and Narayan 2013] have demonstrated that CSLs are useful for the consistency model, but what is the situation for the persistency model?

As we have seen, reasoning about persistency revolves around proving that algorithms maintain *recovery invariants*, which need to hold all the time because crashes can happen any time. And, for now, these recovery invariants appear to be stated as *low-level* and *global* properties that only concern with values of multiple locations. While it is possible to make recovery invariants relatively modular by encapsulating them with the set of locations used by the persistent library in question, it is interesting to see if recovery invariants can be phrased in more *high-level* CSL assertions that allow for *composing* smaller recovery invariants into larger ones, in a fashion similar to composing smaller libraries into larger ones as discussed in §3.

Furthermore, looking at how writes are persisted to the non-volatile memory, it is also not clear if a separation of ownership on the non-volatile memory exists. We need to explore how to cast such persistent properties into the form of CSL assertions, and how to make them compatible with the reasoning of the relaxed memory consistency model. It appears to me that looking at the Px86_{sim} model with *explicit buffers* is not the right way to find the separation structure. Fortunately, several previous works on relaxed memory models [Dang et al. 2019a; Kaiser et al. 2017; Kang et al. 2017; Lahav et al. 2016] provide an *alternative* formulation of such semantics by introducing *per-location histories*. In these models, each location X has a history that is ordered by the consistency order, and each thread maintains its own local *progress* on this order which defines what values the thread would obtain when it reads X, as well as where its writes to X would end up in X's history. Each thread's collections of its progress on all locations constitutes the thread's *local view* of the histories. Then, separation logics can be built on the *separation structure of location histories and thread-local views*. Following this alternative formulation, we can construct both consistency and persistency per-location histories to encode the consistency order and the persistency order,

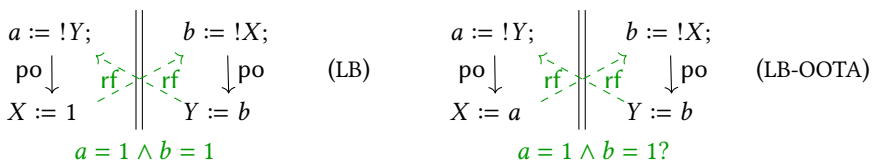
and extend thread-local views to accommodate both of them.[6] Then it probably will be easier to identify the separation structures of consistency and persistency histories and thread-local views and derive a logic for them. This would solve the compatibility problem between consistency and persistency reasoning, and open the possibility of encoding persistent properties in CSL assertions. The remaining specific task would be how to state recovery invariants in such a logic.

Below is a list of specific challenges that I can identify to build a persistency logic for Px86_{sim}. I plan to build the logic in the Iris framework [Jung et al. 2018b], in collaboration with Raad [2020] and colleagues.

- **Handling crash behaviors.** It appears that one would need to model crashes to specify recovery. However, if we only care about recovery invariants which are supposed to hold at every point in the program, we may not need to model crashes at all.
- **Augmenting the semantics with histories and views** so that we can identify the separation structure. This requires encoding the propagation orders of the buffers into histories and views and proving that they remain in agreement through out program executions.
- **Deriving basic, and then more high-level CSL assertions** for histories and views, in such a way that can easily express recovery invariants in a more modular and high-level fashion.
- **Understanding composability of recovery invariants**, to see if we can derive more complex recovery invariants by relating known recovery invariants of the program, and whether such composability is useful in practice.
- **Identifying key examples and performing verifications**. Several algorithms exist without a formal treatment and are non-trivial: persistent lock-free queues [Friedman et al. 2018; Raad and Vafeiadis 2018], hash-maps [Nawab et al. 2017], sets [Zuriel et al. 2019], and transactions algorithms [Kolli et al. 2016; Raad et al. 2019b]. Performing verification for them will validate the usefulness and/or the need of separation logics.

5 PROMISING LOGICS

Most relaxed memory logics [Dang et al. 2019a; Doko and Vafeiadis 2016, 2017; Kaiser et al. 2017; Turon et al. 2014; Vafeiadis and Narayan 2013] have been developed with a restriction that, in the terminology of the C11 axiomatic model, the $(\text{po} \cup \text{rf})^+$ relation must be *acyclic*. That is, these logics can only verify programs that do not have cycles between the program-order relation po and the *reads-from* relation rf. The restriction forbids the following *Load Buffering* (LB) behavior.



Here, we assume that both locations X and Y are initialized with 0. In (LB), it is possible that the right thread reads $b = 1$ from the left thread's write to X, then writes 1 to Y, and the left thread reads from that write to get $a = 1$. As we can see, this behavior forms a cycle in the $(\text{po} \cup \text{rf})^+$ relation, and is allowed in C11. Intuitively, it can be explained by executing the left

[6] In fact, [Raad 2020] and colleagues, even though not using explicit histories and views, are proposing assertions that involve different versions of a location X's values: a volatile version X_v that is the value local to each thread, a store version X_s that is the value in the persistent buffer and is the result of the store order, and a persistent version X_p that is the value in the non-volatile memory and is the result of the persistency order. Then various rules encode how the values are propagated from X_v to X_s to X_p.

thread's write of X before its read of Y. (LB) is observable if a compiler decides to reorder the (apparently independent) write before the read; on ARMv8 [Pulte et al. 2018], it is observable even without compiler transformations.

However, the C11 model is seriously flawed, because by allowing cycles in $(\mathsf{po} \cup \mathsf{rf})^+$, it also allows *out-of-thin-air* (OOTA) behaviors, as demonstrated by the (LB-OOTA) example (see above). In this example, C11 allows that the reads of X and Y (values returned in the local variables a and b) can be 1, even though there is no write of 1 in the program: the value 1 appears out of thin air! To avoid handling such weird behaviors, most logics for RMM simply decided to opt out and not support (LB), which means that they cannot reason about real programs that run on ARM or Power, or programs that are applicable to certain compiler transformations.

Meanwhile, there have been several proposals at the level of memory models to fix the OOTA problem while allowing (LB) [Chakraborty and Vafeiadis 2019; Jeffrey and Riely 2016; Kang et al. 2017; Lee et al. 2020; Pichon-Pharabod and Sewell 2016]. However, there is little work on building high-level program logics for these memory models. To our knowledge, the only published program logic is the SLR logic by Svendsen et al. [2018], built upon the *promising semantics* [Kang et al. 2017]. Unfortunately, SLR's soundness proof is substantially complex in order to handle the behaviors of *promises*—the main ingredient of the promising semantics to model reordering writes earlier. Even so, SLR does not appear to be expressive enough to verify many examples, including those in $\mathbf{RB}_{\mathtt{r1x}}$ [Dang et al. 2019a]. Last but not least, SLR's soundness proof was done only with pen-and-paper, and it does not have a framework to support machine-checked programs verification in the logic.

Below, I discuss the main challenge in reasoning about promises, and propose to explore several ideas that can help overcome the problem.

5.1 The Promising Semantics

Promises are introduced by the operational-style promising semantics [Kang et al. 2017]—a language level memory model to fix C11—to model reordering writes earlier while disallowing OOTA behaviors. Its key idea is that a thread can promise its write earlier (in the program order po) so that its effect can be observed and relied on by other threads. To be consistent, however, the promising thread cannot rely on its own promise, either directly or indirectly through other threads, until it has *fulfill* the promise—that is, eventually performing the write. This means that the promising thread must prove that it can fulfill the promise without help from other threads. This proof is materialized in the promising semantics as a *local certification execution* that comes with the promise. The certification execution is a *sequential* execution *without promising steps* by the promising thread and is separate from the actual execution. Basically, at every step in the actual execution, the thread must show that there exists a certification execution where it runs alone without making more promises (and thus sequentially; all other threads considered frozen) and can reach a point where *all* of its outstanding promises are fulfilled.

For example, in (LB), the left thread can promise the write of 1 to X, because it can show that in any later step it will be able construct a certification execution where it writes 1 to X eventually without help from other threads. In the actual execution, after the promise has happened, the right thread can observe that promise like a normal write, and thus reads 1 from X, then writes that value 1 to Y, which then in turn allows the left thread to proceed to read 1 from Y. Finally, still in the actual execution, the left thread simply writes 1 to X to actually fulfill its promise. (While the (LB) example does not demonstrate it, it is useful to note the fact that the fulfillments in certification executions are not necessarily the same as the fulfillment in the actual execution.) Meanwhile, in (LB-OOTA), the left thread cannot promise to write 1 to X at all, because at any time it cannot show

that it will eventually write 1 without any help from other threads. The same also applies to the right thread for its write to Y.

The formal definition of certification is carefully defined such that it prevents various undesirable behaviors that are not observable in hardware (where hardware prevents such behaviors through syntactic dependencies between instructions), while still allowing reordering of instructions that can be generated by many compiler transformations as well as the hardware themselves. The promising semantics 2.0 [Lee et al. 2020] fine-tunes the definition of certification further to allow for global-analysis-based transformations. However, I will not discuss more details within the scope of this proposal—interested readers can refer to the original papers [Kang et al. 2017; Lee et al. 2020].

5.2 Reasoning About Promises

The main challenge in reasoning about promises is how early a write to a location X can be promised, which determines how early the effect of that write can be visible to other threads. This is particularly important to building program logics, because if we intend to use X as a communication channel between threads, then whatever information we want to communicate through a write of X must be available *at the point of the promise*, not at the point of the fulfillment, such that other threads can access that information. In separation logics, such information can also be extended to *resources*, and then the logic's design question of whether or not we should allow *transferring* resources over *promisable* writes depends on whether the resources can be made ready by the time the promise happens. (SLR [Svendsen et al. 2018] decided to *disallow* resource transfer over promisable writes altogether.) Another design question concerns the *permission* to even make promises: promises are almost as powerful as writes as they represent the intent to insert new events into the history of the location. In fact, from the perspective of other threads, promises are no different from writes. So, in separation logics, similarly to how a thread would need some permission to write to a location, does a thread need to own the permission to make a promise? If so, does that mean that we require the thread to own that permission by the time the promise happens? (SLR does not have such requirement, and relies on certification to justify the lack of it, which is the reason why SLR's soundness proof is substantially complex—see more below.)

Unfortunately, it is non-trivial to determine how early a write can be promised. In principle, a write can be promised as early as possible, as long as in the promising thread's subsequent steps the promise is always fulfillable. And fulfillment relies on the existence of certification executions, which are formally defined with respect to the machine's current *whole* memory, which makes it very *non-modular*, because this means that a thread's promises may only happen after certain actions by other threads. For example, in (LB), the right thread can promise to write 1 to Y, but *only after* the left thread has promised or written 1 to X, because only from that state of the memory can the right thread certify that it can read 1 from X and thus write 1 to Y.

A similar situation can be observed in the following *Message-Passing* (MP) example.

$$\begin{array}{l} X := 1; \\ Y :=_{\texttt{rel}} 1 \end{array} \;\Big\|\; \begin{array}{l} a := !^{\texttt{acq}} Y; \\ \texttt{if}\ a = 1\ \texttt{then} \\ \quad X := 2 \end{array} \tag{MP}$$

Here, from the perspective of a separation logic, we are using a release-acquire synchronization on Y to transfer (message-passing) the ownership of X from the left thread to the right thread so that the right thread can write to X. The question of interest is how early the write of 2 to X by the right thread can be promised. By certification, the right thread can promise to write 2 to X *as soon as* the left thread has finished writing 1 to Y, that is, even before the read of Y and the conditional

are executed. From the perspective of a logic designer, should we model that the right thread has acquired the resources of X right at the time the promise to X is made? This should be sound because the promise can only be introduced by the semantics with a certification which justifies that the right thread must have been able to access X at that time. But proving the soundness of such a scheme is non-trivial, because we have to handle not only the actual execution, but also the certification executions.

In general, if one wants to allow resource transfer over promisable writes, it is unclear where the resources should be and when they should be transferred, due to the too flexible semantics nature of promises. This is why the SLR logic [Svendsen et al. 2018] forbids transferring resources over promisable writes. At first, this appeared to us as too restrictive, because we did not see a clear way to verify $\mathbf{RB_{rlx}}$ examples (like `Arc`) in such a logic. However, after a long time *failing* to find a general resource transfer scheme for promisable writes, I believe a compromise can be made by disallowing resource transfer over promisable writes in certain cases, while allowing it in other synchronization schemes (for example, release-acquire synchronization with relaxed accesses and fences). There is also a possibility to adjust the verifications in $\mathbf{RB_{rlx}}$ to fit into this compromise.

SLR also made another design choice: to completely hide reasoning about promises in the logic. That is, users of SLR logic would prove Hoare triples in the *non-promising* fragment of the semantics, where new promises cannot be made. Then, the Herculean soundness proof of SLR guarantees that any such triple that holds in the non-promising semantics also holds in the promising semantics. This is only possible thanks to the various restrictions enforced by the reasoning rules of SLR. While this is an interesting design choice that keep the logic's reasoning very clean, it is unclear how it would fit into the compromise mentioned above, where the logic would be less restrictive.

Meanwhile, there is also a development of the promising semantics at hardware level for ARM [Pulte et al. 2019]. The promising ARM semantics also seems to be a good target for a logic, as it encodes more *syntactic dependencies* from ARM, making the behaviors of promises more constrained. In particular, due to the syntactic dependencies, in many cases, the order in which writes can be promised earlier actually agrees with the order in which writes should happen. This means that in many cases, the behaviors of promises follow the non-promising behaviors of writes, so there is a possibility to start with reasoning in the non-promising machine and replay that reasoning in the promising machine for promises, thus hiding reasoning about promises in the logic, like in SLR. This is an interesting direction that can be explored in collaboration with Pulte et al. [2020].

Having considered the various aspects mentioned above, I propose to explore the following research questions.

- Are the restrictions of SLR the right ones? Can we relaxed them so as to be able to verify more complex program verifications, like those in $\mathbf{RB_{rlx}}$?
- Can we achieve the soundness proof of SLR in a machine-checked framework like Iris? This would require a version of Iris that supports *transfinite step-indexing*, as SLR's soundness proof need to reasoning about both the actual execution and the certification executions.
- Would the more constrained behaviors of the promising ARM semantics give us a simpler soundness proof, and make its formalization more conceivable in Iris?
- Promises also have the power of *predicting* future executions, for example, in (MP), the promise of writing 2 to X by the right thread proves that the thread must take the true branch of the conditional. Can such power be exploited for reasoning?

Timeline	Action Point		
	StrongRLX (§3)	**PerSL** (§4)	**ProSL** (§5)
2020 June - July	Start exploration, in collaboration with Mansky [2020] and [Kang 2020]		
2020 August - October	Planned internship		
2020 November	Continue with project	Start exploration if **StrongRLX** fails, in collaboration with [Raad 2020]	Start exploration if **PerSL** fails, in collaboration with [Pulte et al. 2020]
2020 December - 2021 June	Continue with the chosen project(s), and prepare for publication(s)		
2021 July - August	Thesis writing		

Fig. 4. Suggested timeline for thesis completion

6 TIMELINE FOR THESIS COMPLETION

I propose the timeline in Figure 4 to complete the thesis, with the options of three projects: stronger specifications for RMM (**StrongRLX**, §3), persistency logics (**PerSL**, §4), and promising logics (**ProSL**, §5).

REFERENCES

Martín Abadi and Leslie Lamport. 1991. The existence of refinement mappings. *Theoretical Computer Science* 82, 2 (1991), 253 – 284. https://doi.org/10.1016/0304-3975(91)90224-P

Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic foundations for typed assembly languages. *TOPLAS* 32, 3 (2010), 1–67.

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *POPL*. 55–66.

Hans-J. Boehm and Dhruva R. Chakrabarti. 2016. Persistence Programming Models for Non-Volatile Memory. ISMM (2016). https://doi.org/10.1145/3241624.2926704

Hans-J. Boehm and Brian Demsky. 2014. Outlawing ghosts: Avoiding out-of-thin-air results. In *MSPC*.

John Boyland. 2003. Checking interference with fractional permissions. In *SAS (LNCS)*. https://doi.org/10.1007/3-540-44898-5_4

Stephen Brookes. 2007. A Semantics for Concurrent Separation Logic. *Theoretical Computer Science* 375, 1–3 (2007). https://doi.org/10.1016/j.tcs.2006.12.034

Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding Thin-Air Reads with Event Structures. POPL (2019). https://doi.org/10.1145/3290383

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP*, Richard Jones (Ed.).

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019a. RustBelt Meets Relaxed Memory. *Proc. ACM Program. Lang.* POPL (2019). https://doi.org/10.1145/3371102

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019b. RustBelt meets relaxed memory – Artifact. https://doi.org/10.5281/zenodo.3539237 Latest version available at http://plv.mpi-sws.org/rustbelt/rbrlx/.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent abstract predicates. In *ECOOP (LNCS)*. https://doi.org/10.1007/978-3-642-14107-2_24

Marko Doko and Viktor Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *VMCAI (LNCS)*. Springer, 413–430.

Marko Doko and Viktor Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *ESOP*.

Xinyu Feng. 2009. Local Rely-Guarantee Reasoning. In *POPL*. https://doi.org/10.1145/1594834.1480922

Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *ESOP*.

Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-Free Queue for Non-Volatile Memory. In *PPoPP*. https://doi.org/10.1145/3178487.3178490

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *CONCUR*.

Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-Free Regions. PLDI (2018). https://doi.org/10.1145/3296979.3192367

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. 2007. Local Reasoning for Storable Locks and Threads. In *APLAS* (Singapore). 19–37.

Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. 2015. Modular Verification of Concurrency-Aware Linearizability. In *DISC*. https://doi.org/10.1007/978-3-662-48653-5_25

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* (1990). https://doi.org/10.1145/78969.78972

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *ESOP* (Budapest, Hungary). 353–367.

Intel. 2014. Intel architecture instruction set extensions programming reference. (2014). https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf

Intel. 2019. Intel 64 and IA-32 Architectures Software Developer's Manual (Combined Volumes). (2019). https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

Joseph Izraelevitz, Hammurabi Mendes, and Michael Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing (DISC)*. https://doi.org/10.1007/978-3-662-53426-7_23

Bart Jacobs and Frank Piessens. 2011. Expressive Modular Fine-Grained Concurrency Specification. In *POPL*. https://doi.org/10.1145/1926385.1926417

Alan Jeffrey and James Riely. 2016. On Thin Air Reads Towards an Event Structures Model of Relaxed Memory. In *LICS*. https://doi.org/10.1145/2933575.2934536

Jonas Braband Jensen and Lars Birkedal. 2012. Fictional Separation Logic. In *ESOP (LNCS)*. https://doi.org/10.1007/978-3-642-28869-2_19

Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient Persist Barriers for Multicores. In *MICRO-48*. https://doi.org/10.1145/2830772.2830805

Jacques-Henri Jourdan. 2018. Insufficient synchronization in Arc::get_mut. Rust issue #51780, https://github.com/rust-lang/rust/issues/51780.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the Foundations of the Rust Programming Language. *PACMPL* 2, POPL, Article 66 (2018).

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *Journal of Functional Programming* 28, e20 (Nov. 2018), 1–73. https://doi.org/10.1017/S0956796818000151

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The Future is Ours: Prophecy Variables in Separation Logic. *Proc. ACM Program. Lang.* 4, POPL (2020). https://doi.org/10.1145/3371113

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*. https://doi.org/10.1145/2676726.2676980

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 *ECOOP (LIPIcs)*. 17:1–17:29.

Jeehoon Kang. 2020. Personal communication.

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-memory Concurrency. In *POPL* (Paris, France). ACM, 175–189.

Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. https://doc.rust-lang.org/stable/book/2018-edition/

Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-Level Persistency. *SIGARCH Comput. Archit. News* ISCA (2017). https://doi.org/10.1145/3140659.3080229

Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *ASPLOS*. https://doi.org/10.1145/2872362.2872381

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *PACMPL* 2, ICFP, Article 77 (2018).

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL*. https://doi.org/10.1145/3009837.3009855

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming Release-Acquire Consistency. POPL (2016). https://doi.org/10.1145/2914770.2837643

Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *ICALP*.

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *PLDI*.

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691.

Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: Global Optimizations in Relaxed Memory Concurrency. PLDI (2020).

William Mansky. 2020. Personal communication.

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating state transition systems for fine-grained concurrent resources. In *ESOP (LNCS)*. https://doi.org/10.1007/978-3-642-54833-8_16

Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dalí: A Periodically Persistent Hash Map. In *DISC*. https://doi.org/10.4230/LIPIcs.DISC.2017.37

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theoretical Computer Science* 375, 1-3 (2007). https://doi.org/10.1016/j.tcs.2006.12.035

Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. *SIGARCH Comput. Archit. News* ISCA (2014). https://doi.org/10.1145/2678373.2665712

Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-Air Executions. In *POPL*. https://doi.org/10.1145/2837614.2837616

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. POPL (2018). https://doi.org/10.1145/3158107

Christopher Pulte, Jean Pichon-Pharabod, and Jeehoon Kang. 2020. Personal communication.

Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: A Simpler and Faster Operational Concurrency Model. In *PLDI*. https://doi.org/10.1145/3314221.3314624

Azalea Raad. 2020. Personal communication.

Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. 2019a. On Library Correctness under Weak Memory Consistency: Specifying and Verifying Concurrent Libraries under Declarative Consistency Models. *Proc. ACM Program. Lang.* POPL (2019). https://doi.org/10.1145/3290381

Azalea Raad and Viktor Vafeiadis. 2018. Persistence Semantics for Weak Memory: Integrating Epoch Persistency with the TSO Memory Model. OOPSLA (2018). https://doi.org/10.1145/3276507

Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020. Persistency Semantics of the Intel-X86 Architecture. *Proc. ACM Program. Lang.* 4, POPL (2020). https://doi.org/10.1145/3371079

Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019b. Weak Persistency Semantics from the Ground up: Formalising the Persistency Semantics of ARMv8 and Transactional Models. *Proc. ACM Program. Lang.* OOPSLA (2019). https://doi.org/10.1145/3360561

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer's Model for X86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. https://doi.org/10.1145/1785414.1785443

Kasper Svendsen and Lars Birkedal. 2014. Impredicative concurrent abstract predicates. In *ESOP (LNCS)*. https://doi.org/10.1007/978-3-642-54833-8_9

Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A Separation Logic for a Promising Semantics. In *ESOP*.

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*. https://doi.org/10.1145/2500365.2500600

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. In *OOPSLA*. ACM, 691–707.

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In *OOPSLA*.

Viktor Vafeiadis and Matthew J. Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR*. 256–271.

Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient Lock-Free Durable Sets. OOPSLA (2019). https://doi.org/10.1145/3360554