A Verified High-Performance Composable Object Library for Remote Direct Memory Access

GUILLAUME AMBAL*, Imperial College London, UK
GEORGE HODGKINS*, University of Colorado, Boulder, USA
MARK MADLER, University of Colorado, Boulder, USA
GREGORY CHOCKLER, University of Surrey, UK
BRIJESH DONGOL, University of Surrey, UK
JOSEPH IZRAELEVITZ, University of Colorado, Boulder, USA
AZALEA RAAD, Imperial College London, UK
VIKTOR VAFEIADIS, MPI-SWS, Germany

Remote Direct Memory Access (RDMA) is a memory technology that allows remote devices to directly write to and read from each other's memory, bypassing components such as the CPU and operating system. This enables low-latency high-throughput networking, as required for many modern data centres, HPC applications and AI/ML workloads. However, baseline RDMA comprises a highly permissive weak memory model that is difficult to use in practice and has only recently been formalised.

In this paper, we introduce the Library of Composable Objects (LOCO), a formally verified library for building multi-node objects on RDMA, filling the gap between shared memory and distributed system programming. LOCO objects are well-encapsulated and take advantage of the strong locality and the weak consistency characteristics of RDMA. They have performance comparable to custom RDMA systems (e.g. distributed maps), but with a far simpler programming model amenable to formal proofs of correctness.

To support verification, we develop a novel modular declarative verification framework, called Mowgli, that is flexible enough to model multinode objects and is independent of a memory consistency model. We instantiate Mowgli with the RDMA memory model, and use it to verify correctness of LOCO libraries.

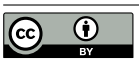
CCS Concepts: • **Theory of computation → Axiomatic semantics**; **Distributed computing models**; • **Software and its engineering → Distributed programming languages**.

Additional Key Words and Phrases: RDMA, Distributed Computing, Declarative Semantics, Verification

ACM Reference Format:
Guillaume Ambal, George Hodgkins, Mark Madler, Gregory Chockler, Brijesh Dongol, Joseph Izraelevitz, Azalea Raad, and Viktor Vafeiadis. 2026. A Verified High-Performance Composable Object Library for Remote Direct Memory Access. *Proc. ACM Program. Lang.* 10, POPL, Article 71 (January 2026), 62 pages. https://doi.org/10.1145/3776713

*co-first authors.

Authors' Contact Information: Guillaume Ambal, Imperial College London, UK, g.ambal@imperial.ac.uk; George Hodgkins, University of Colorado, Boulder, USA, George.Hodgkins@colorado.edu; Mark Madler, University of Colorado, Boulder, USA, Mark.Madler@colorado.edu; Gregory Chockler, University of Surrey, UK, g.chockler@surrey.ac.uk; Brijesh Dongol, University of Surrey, UK, b.dongol@surrey.ac.uk; Joseph Izraelevitz, University of Colorado, Boulder, USA, Joseph.Izraelevitz@colorado.edu; Azalea Raad, Imperial College London, UK, azalea.raad@imperial.ac.uk; Viktor Vafeiadis, MPI-SWS, Germany, viktor@mpi-sws.org.



ACM 2475-1421/2026/1-ART71
https://doi.org/10.1145/3776713

1 Introduction

The *remote direct memory access* (RDMA) protocol provides a load/store interface, allowing a machine to access the memory of a remote machine across a network without communicating with the remote processor. The memory accesses are performed directly by the *network interface card* (NIC), bypassing the software networking stack on both ends of the connection. As such, RDMA achieves low-latency, high-throughput communication, making it a key technology in many production-grade data centres such as those at Microsoft [Zhu et al. 2015], Google [Lu et al. 2018], Alibaba [Wang et al. 2023b], and Meta [Gangidi et al. 2024].

Despite its memory-like interface, RDMA is a hardware-accelerated networking protocol, and has traditionally been programmed as such—not as shared memory. This has resulted in a very weak memory model with out-of-order behaviours visible even in a sequential setting [Ambal et al. 2024]. Consider, for example, the following program, where all memories are zero-initialised.

$\overline{z} := x$; // RDMA put: write the value of local variable x to remote location z
$x := 1$ // update local variable x to 1

Somewhat counterintuitively, this program can result in z getting the value 1, with the following execution steps: (1) the put instruction ($\overline{z} := x$) is offloaded to the NIC; (2) the CPU executes $x := 1$ updating the value of x in the local memory; and (3) the NIC executes the put instruction, fetching the *new* value of x from local memory before performing the remote write.

Since programming RDMA directly is challenging, prior work has developed custom RDMA libraries. Most existing libraries are monolithic: they encapsulate a useful distributed protocol (such as consensus [Aguilera et al. 2020] or distributed storage [Dragojević et al. 2014; Wang et al. 2022]) as a single, global entity—not one that can be reused by other RDMA libraries. Some other libraries (e.g. [Cai et al. 2018; Wang et al. 2020]) provide a simple high-level memory abstraction that hides all the complexities of a highly non-uniform, weakly consistent network memory, but also loses a lot of the performance that can be achieved by knowing the system layout [Liu and Mellor-Crummey 2014; Majo and Gross 2017; Tang et al. 2013]. Other intermediate layers, such as MPI [Message Passing Interface Forum 2023] or NCCL [NVIDIA Corporation 2020] are designed explicitly for networks and present a message passing interface that is ideal for embarrassingly parallel or task-oriented workflows, but ill-suited for irregular and data-dependent workloads, such as data stores or stateful transactional systems, for which shared-memory solutions excel [Liu et al. 2021]. Although these library implementations are impressive engineering artefacts and have often been carefully tuned to achieve very good performance, they are almost impossible to verify formally due to their lack of modularity.

In this paper, we argue for a new way for programming RDMA applications—and more generally systems with non-uniform weakly consistent memories—with *flexible* libraries that can expose the non-uniform memory aspects and that support formal verification. Key to our approach is *composability*—namely, the ability to put together smaller/simpler objects to build larger ones—and this composability is reflected both in the design and implementation of our library as well as in the formal proofs about its correctness.

LOCO. As a first contribution, we introduce the *Library of Composable Objects (LOCO).* A LOCO object is a concurrent object as in Herlihy and Wing [1990a], exposing a collection of methods, but storing its state in a distributed fashion across all participating nodes. Familiar examples include cross-node locks, barriers, queues, and maps. LOCO objects provide encapsulation and can be composed together to build other LOCO objects. We define objects encapsulating the underlying RDMA operations and the local CPU instructions, and use them to build intermediate objects, such as ring buffers, which in turn are used to build larger objects, such as a key-value store (see Fig. 1).

The source code for LOCO is available on Github [⌥]. Additionally, a previous preprint version of this paper focused on the LOCO library was published on arXiv [Hodgkins et al. 2025].

For concreteness, we implement and verify LOCO objects over RDMA^{TSO} (which combines an RDMA networking fabric with Intel x86-TSO nodes), making use of an existing formalisation by Ambal et al. [2024]. RDMA^{TSO} is, however, too low-level for our purposes: it does not provide a compositional way to wait for RDMA operations to complete, making it impossible to encapsulate it as a LOCO object. For this reason, we introduce RDMA^{WAIT}, a thin layer of abstraction over RDMA^{TSO}, that attaches identifiers to RDMA operations and allows threads to perform a `Wait` operation to wait for all RDMA operations with a given identifier to finish executing. To attain good performance, the practical implementation of the `Wait` operation in LOCO is quite involved. Nevertheless, we prove the correctness of a simplified version over the underlying RDMA^{TSO} model.

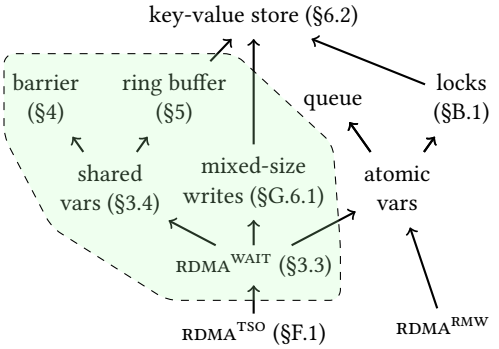


Fig. 1. LOCO libraries with their dependencies

MOWGLI. As a second contribution, we introduce Mowgli (MOdular Weak Graph-based LIbraries), a generic, modular framework for modelling and verifying weak libraries. Mowgli is *generic* in that it makes no assumptions about the underlying memory model (e.g. RDMA or TSO) in its core theory; and it is *modular* in that it allows proof decomposition at library interfaces and reasoning about individual components without referring to the internals of other components.

We instantiate Mowgli with RDMA^{WAIT} and establish the correctness of all the LOCO libraries that do not depend on atomic read-modify-write (RMW) RDMA operations because the latter are not covered by the existing RDMA^{TSO} model. The verified libraries are highlighted in Fig. 1.

Mowgli represents program executions as graphs, whose nodes are called *events* and represent either a simple operation like a read or a write, or a more complex operation such as a method call. Following the declarative approach of Raad et al. [2019] and Stefanesco et al. [2024], we specify each concurrent object with a set of axioms (i.e., consistency predicates) over events. As we shall see in §2.3, however, events are too coarse-grained to model the intricate synchronisation guarantees of RDMA operations.

We therefore introduce the novel notion of a *subevent*, allowing one to split complex library operations into multiple subevents, each with a different *stamp* (representing, e.g., the node affected by the subevent). Stamps are meta-categories of behaviours, shared by all libraries, and are independent from programs. Stamps are then used to induce ordering among (sub)events. Within a thread, they are used to define the *preserved program order* (ppo) [Alglave et al. 2014], which relates (sub)events executed by a thread that may not be reordered. Across threads and nodes, stamps are used to define the *synchronisation order* (so) [Dongol et al. 2018] between methods calls of the same library. Together ppo and so are used to define the happens-before relation.

Our main result supporting modular proofs in Mowgli is a new locality result that decomposes proving correctness of a system into proofs about the correctness of its individual components. This is akin to the notion of compositionality for linearisability [Herlihy and Wing 1990a], but generalised to a partially ordered setting. In our verification of LOCO, this allows us to verify a library, then use the *specification* of the library in any program that uses the library. Moreover, we show that our locality result supports both *horizontal composition*, where a library is used within a

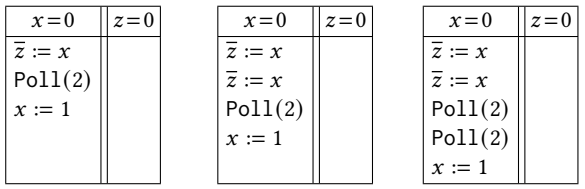


Fig. 2. Polling under RDMA^{TSO}

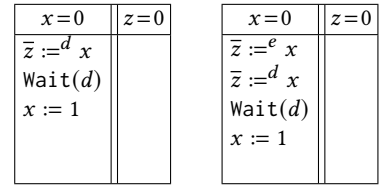


Fig. 3. Waiting under RDMA^{WAIT}

client program, and *vertical composition*, where a library is developed from other libraries via a series of abstractions.

Contributions. In summary, we make the following contributions:

- We define a new consistency model, RDMA^{WAIT}, that supports a `Wait` operation that allows CPUs to wait for the confirmation (by the NIC) for a *specific* group of remote operations. We verify the correctness of the RDMA^{WAIT} implementation over the existing RDMA^{TSO} model.
- We develop LOCO [], a flexible, modular object library for RDMA, and demonstrate its compositionality by using simpler objects to build more advanced objects: e.g., a barrier, a ring buffer, a linearisable key-value store, a transactional locking scheme, and a distributed DC/DC converter.
- We introduce a new modular formal framework, Mowgli, for specifying and verifying concurrent libraries over weakly consistent memory and distributed architectures.
- We instantiate Mowgli to verify correctness of the aforementioned LOCO libraries.
- We benchmark LOCO's barrier and ring buffer objects and show that they outperform the highly-tuned OpenMPI implementations of the same objects.

2 Overview of LOCO and MOWGLI

In this section, we provide an informal, more detailed overview of LOCO and Mowgli. We present LOCO's base memory model, RDMA^{WAIT}, in §2.1, then discuss the key libraries that we consider. In §2.3, we provide an overview of our Mowgli verification framework.

2.1 The RDMA^{WAIT} Memory Model

We start by informally describing LOCO's base memory model, RDMA^{WAIT}, and contrast it to RDMA^{TSO} [Ambal et al. 2024] via a set of simple examples. Both models provide `put` operations ($\overline{x} := y$) for writing to remote memory and `get` operations ($y := \overline{x}$) for reading from remote memory, which are executed asynchronously. The models differ in how a thread can wait for these asynchronous operations to terminate.

In RDMA^{TSO}, waiting is achieved with the `Poll` primitive. Consider the programs in Fig. 2, which comprise two nodes, with a variable x in node 1 and a variable z in node 2. In the first program, Fig. 2a, node 1 comprises a single thread that first puts the value of x to the remote location z (located in node 2), and then polls node 2, which causes the thread to wait until the put has been executed, and finally updates x to 1. This means that the final value of z is 0, and not 1. Note that in the absence of the `Poll` operation, the final outcome $z = 1$ would be permitted since the instruction $\overline{z} := x$ could simply be offloaded to the NIC, followed by the update of x to 1. When $\overline{z} := x$ is later executed by the NIC, it will load the value 1 for x.

Synchronisation via `Poll` is however brittle, and sensitive to the number of instructions occurring before the `Poll`. For example, as shown in Fig. 2b, the final outcome $z = 1$ is once again permitted because the `Poll` only waits for the earliest unpolled operation to be executed at node 2. In particular,

although `Poll` does wait for the first put instruction, the second put may be offloaded to the NIC and the local write $x := 1$ executed before the second put ($\overline{z} := x$) is executed. This weak behaviour is also allowed if we replace the first operation with *any* RDMA operation, even unrelated to locations x and z. This demonstrates that RDMA$^{\text{TSO}}$ programs are *not compositional*: we cannot reason about a property (e.g. the final value of z) by focusing only on the part of the program that seems relevant; only monolithic analysis of the full program is possible. To prevent the weak behaviour of Fig. 2b, one must add a second `Poll` operation as shown in Fig. 2c.

In RDMA$^{\text{WAIT}}$, synchronisation is performed with the `Wait` operation. RDMA operations are associated with a work identifier, e.g. d in Fig. 3a, which can be waited upon with a `Wait` operation. Thus, unlike `Poll`, which waits for the first unpolled operation, RDMA$^{\text{WAIT}}$ can wait for a specific put or get operation. This improves robustness since the `Wait` is independent of the number of instructions that have been executed by each thread. For example, in Fig. 3b, the `Wait` can target the *second* put instruction using the work identifier d and exclude the unintended outcome $z = 1$.

While `Wait` makes targeting a remote operation easier, it does not provide more synchronisation guarantees than the `Poll` operation. Waiting for a put operation ($\overline{z} := x$) only guarantees that the local value of x has been read, not that the remote location z has been modified. Thus, as shown in Fig. 4, the store buffering behaviour across nodes is possible even if we wait for every remote operation. In contrast, waiting for a get operation ($x := \overline{z}$) does guarantee it has fully completed, i.e. that z has been read and x modified. This can be exploited to prevent the store buffering behaviour. RDMA ordering rules ensure that later gets execute after previous puts towards the same remote node. Thus, waiting for a (seemingly unrelated) get operation can be used to ascertain the completion of previous remote writes.

$y=0$	$x=0$
$\overline{x} :=^{d} 1$	$\overline{y} :=^{e} 1$
`Wait`(d)	`Wait`(e)
$a := y$	$b := x$

$(a, b) = (0, 0)$ ✓

$y, w=0$	$x, z=0$
$\overline{x} := 1$	$\overline{y} := 1$
$c :=^{d} \overline{z}$	$d :=^{e} \overline{w}$
`Wait`(d)	`Wait`(e)
$a := y$	$b := x$

$(a, b) = (0, 0)$ ✗

Fig. 4. Preventing RDMA store buffering

We present the formal definitions of RDMA$^{\text{WAIT}}$ in §3.3 using a declarative style. Although, like RDMA$^{\text{TSO}}$, it is also possible to derive an equivalent operational model, we elide these details since the proof technique that we use (see §2.3) directly uses the declarative semantics.

Note that the actual implementation of `Wait` in LOCO is non-trivial, relying on a highly optimised code path to track outstanding operations and match them to an associated `Wait`. This extension to the RDMA interface is described within Section 2.2, with a more complete treatment in Appendix C.

2.2 LOCO Libraries

LOCO provides a set of commonly used distributed objects, which we call *channels*, built on top of RDMA$^{\text{WAIT}}$. Channels are *named* and *composable*. To communicate over a channel, each participating node constructs a local channel object, or *channel endpoint*, with the same name. Each channel endpoint allocates zero or more named local regions of network memory when constructed, and delivers the metadata necessary to access these local memory regions to the other endpoints during the setup process.

Channels make it easy to develop RDMA applications and prove their correctness, for minimal performance loss. A LOCO application will usually consist of many channels (objects) of many different channel types (classes). In addition, each channel can itself instantiate member sub-channels. For instance, a key-value store might include several mutexes as sub-channels to synchronise access to its contents.

Shared Variable Library (sv, §3.4). One of the most basic components of LOCO is the *shared variable* library. Each shared variable is replicated across all (participating) nodes in the network and supports $\texttt{Write}_{\text{sv}}$ and $\texttt{Read}_{\text{sv}}$ operations, which only access the *local* copy of the variable. Any updates to the variable may be pushed to the other replicas by the modifying node with a $\texttt{Bcast}_{\text{sv}}$ operation.[1] We provide examples in §2.3, Fig. 9.

$y = 0$	$x = 0$
$\overline{x} := 1$	$\overline{y} := 1$
$\texttt{GF}_{\text{sv}}(2)$	$\texttt{GF}_{\text{sv}}(1)$
$a := y$	$b := x$

$(a, b) = (0, 0)$ ✗

Fig. 5. Using $\texttt{GF}_{\text{sv}}$

The shared variable library also provides a mechanism for synchronising different nodes using a *global fence* ($\texttt{GF}_{\text{sv}}$) operation. $\texttt{GF}_{\text{sv}}$ takes the node(s) on which the fence should be performed as a parameter and causes the executing thread to wait until all prior operations executed by the thread towards the given nodes have fully completed. This is stronger than using the `Wait` primitive, as the global fence also ensures the remote write parts have completed. An example program using a $\texttt{GF}_{\text{sv}}$ is the store buffering setting given in Fig. 5, which disallows the final outcome $(a, b) = (0, 0)$, but allows all other combinations for a and b with values from $\{0, 1\}$. As can be guessed from the similarity with Fig. 4, this global fence can be implemented by submitting get operations and waiting for them.

Barrier Library (BAL, §4). A commonly used object in distributed systems is a barrier, which provides a stronger synchronisation guarantee than global fences. All threads synchronising on a barrier must finish their operations before execution continues. For example, consider the program in Fig. 6, which only allows the final outcome $(a, b) = (1, 1)$ and forbids all other outcomes. Here, nodes 1 and 2 synchronise on the barrier z, and hence nodes 1 and 2 both wait until both writes to x and y have completed.

$y = 0$	$x = 0$
$\overline{x} := 1$	$\overline{y} := 1$
$\texttt{BAR}_{\text{BAL}}(z)$	$\texttt{BAR}_{\text{BAL}}(z)$
$a := y$	$b := x$

$(a, b) = (1, 1)$ ✓

Fig. 6. Using $\texttt{BAR}_{\text{BAL}}$

Ring Buffer Library (RBL, §5). Similarly useful is a ring buffer, which allows one to develop producer-consumer systems. LOCO's ring buffer supports a one-to-many broadcast, and is the most sophisticated of the libraries that we consider.

Mixed-Size Writes (MSW, §G.6.1). The final library we consider is the mixed-size write library, which allows safe transmission of data spanning multiple words. Here, due to the asynchrony between the CPU and the NIC, it is possible for corrupted data to be transmitted that does not correspond to any write performed by the CPU. There are multiple solutions to this problem; we consider a simple solution that transmits a hash alongside the data.

LOCO API Example. As an example of the LOCO C++ API, Fig. 7 shows our implementation of a barrier object, based on Gupta et al. [2002]. The class uses an array (`arr`) of shared variables as a sub-object [Jha et al. 2019, 2017], demonstrating composition. As with a traditional shared memory barrier, it is used to synchronise all participants at a certain point in execution. For each use of the

```
1  class barrier : public loco::channel {
2    unsigned count;
3    loco::var_array<unsigned> arr;
4    public:
5    void waiting() {
6      // complete outstanding RDMA ops
7      loco::global_fence();
8      count++; // increment our counter
9      arr[loco::my_node()].store(count);
10     arr[loco::my_node()].push_broadcast
           (); //and push
11     bool waiting = true;
12     while(waiting){  // wait for others
13       waiting = false; // to match
14       for (auto& i : arr) {
15         if (i.load() < count){
16           waiting = true;
17           break;}
18 } } } };
```

Fig. 7. Complete C++ code for the LOCO barrier

[1] It is also possible for replicas to pull the new value from a source node when a shared variable is modified, but we do not model this aspect because it is not used in the libraries we consider. Moreover, LOCO also defines a stronger form of a shared variable called an *owned variable*, which provides a mechanism for defining a variable's owner that provides a single authoritative version of the variable (describing its true value), defining a single-writer multi-reader register.

barrier, participants increment their local `count` variable, then broadcast the new value to others using their index in the array. They then wait locally, leaving the barrier only when all participants have a count in the array not less than their own. This code is a near-complete implementation of a single-threaded barrier in LOCO, missing only a boilerplate constructor.

Implementing RDMA^{WAIT}. In general, RDMA operations are assigned a unique ID at initialisation. Subsequent queries to a corresponding *completion queue* (i.e. via the `Poll` operation) indicate the oldest ID that has been received at the remote node and acknowledged. As mentioned in Section 2.1, this default system results in non-local effects.

In contrast, LOCO's backend allows for a practical implementation of RDMA^{WAIT} with a high-performance and composable system for tracking RDMA operations. LOCO uses a dedicated *polling thread* to query the completion queue and notify the application of tracked RDMA operations. If the application wishes to monitor the progress of a single RDMA operation (or a set of them, e.g. for a broadcast to all remote nodes), it creates a special `ack_key` object with the associated operation IDs. In RDMA^{WAIT}, `ack_key` objects are abstracted by work identifiers (see §2.1). When all associated IDs have been dequeued by the polling thread, the `ack_key` object is marked completed. The `ack_key` object exports methods to check its status, i.e. the `Wait` operation simply looks for a completed status. Communication between the application and the polling thread for outstanding operation IDs is managed via a single-writer, multiple-reader lock-free queue [Morrison and Afek 2013]. A full description of this system can be found in Appendix C.

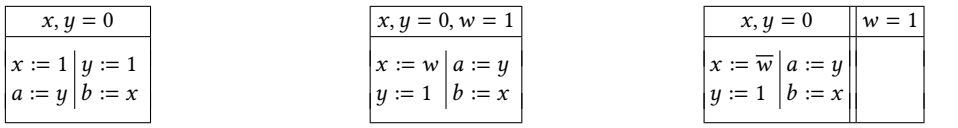
Additional LOCO Libraries. In addition to the proven libraries that are the focus of this paper, LOCO [] [Hodgkins et al. 2025] contains a number of additional objects that rely on an RDMA read-modify-write primitive currently missing from the formalisation provided in RDMA^{TSO}. These include an atomic variable library for accessing these operations, a set of locks (both ticket and test-and-set with optional local flat-combining [Hendler et al. 2010]), and a shared FiFo queue porting the cyclic ring queue [Morrison and Afek 2013]. We intend to fully prove the correctness of these libraries in future work.

LOCO-Based Applications. As mentioned earlier, LOCO enables one to quickly build distributed applications. We demonstrate this by using LOCO to construct a linearisable key-value store (§6.2), a transactional locking scheme (§B.1), and a distributed DC/DC converter (§B.2). Additional obvious targets for LOCO include distributed shared memories [Kaxiras et al. 2015; Keleher et al. 1994], distributed communication collectives [Graham et al. 2006], and other HPC communication library backends (e.g. global arrays [Nieplocha et al. 1994; Zheng et al. 2014]).

2.3 Towards a Modular Verification Framework for LOCO

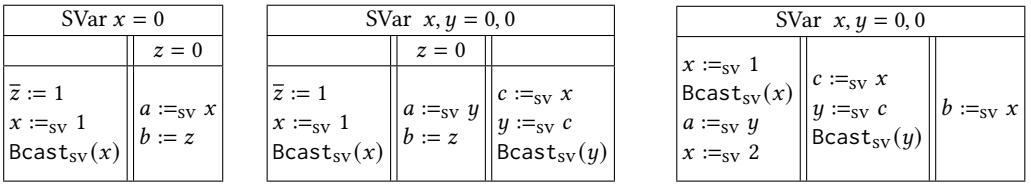
To support reasoning about LOCO libraries, we develop a modular verification framework for RDMA^{WAIT} programs. Our point of departure is the Yacovet framework [Raad et al. 2019; Stefanesco et al. 2024] that was used to reason about weak shared memory *within* a single node. Yacovet, however, is not expressive enough to model RDMA^{WAIT} programs, and so we need to develop a framework that can take into account both sources of weak consistency: shared-memory concurrency (TSO) and distribution (RDMA). This poses three main challenges.

Lack of Causality. RDMA^{WAIT} assumes the TSO memory model [Alglave et al. 2014; Owens et al. 2009] for each CPU within each node. This means that well-known effects such as store buffering (see Fig. 8a) are possible, where both reads in the two threads read from the initial state. Despite this weakness, TSO guarantees causal consistency, i.e. message passing (see Fig. 8b), where the right thread reading the new value 1 for y guarantees that it also reads 1 for x. Formally, this is due to a relation known as *preserved program order* (ppo) between the read of w, the write of this value to x,



(a) Store buffering (b) Message passing (c) Remote message passing

Fig. 8. TSO effects of RDMA$^{\text{WAIT}}$



(a) Broadcast MP (b) Three-node broadcast (c) Broadcast dependency cycle

Fig. 9. Broadcast synchronisation

and the write to y. However, under RDMA$^{\text{WAIT}}$, when interacting with the NIC, causal consistency is no longer guaranteed (see Fig. 8c). This leads to our first modelling challenge: RDMA$^{\text{WAIT}}$ has a much weaker ppo relation than TSO [Alglave et al. 2014]. Here, compositionality is critical to ensure proofs for scalability; we offer this through our locality result (Theorem 3.14).

Fine-Grained Synchronisation. A second challenge in specifying RDMA libraries is that the *same* method call may interact with different library methods in different ways. To make this problem concrete, consider a version of message passing in Fig. 9a, where node 1 updates the remote variable z (located in node 2), and then broadcasts a new value of a shared variable x to signal that the remote value has changed. In Fig. 9a, when node 2 sees the new value of x, it means that the (earlier) write to z must have also taken effect. To represent this, we require that $\overline{z} := 1$ happens before $\texttt{Bcast}_{\text{SV}}(x)$ and that $\texttt{Bcast}_{\text{SV}}(x)$ happens before $a :=_{\text{SV}} x$. These orders *must* be part of the declarative semantics, in some shape or form, to disallow the behaviour $(a, b) = (1, 0)$.

However, naively specifying broadcast in this way is problematic. Consider the example in Fig. 9b, where node 1 behaves as before, but the "signal variable" x is picked up by node 3 and a new signal using y is broadcast by node 3. This time, when node 2 receives the signal on y (i.e. $a = 1$), there is actually no guarantee that the write on z has completed. The outcome $(a, b) = (1, 0)$ is allowed, as communication between each pair of nodes is independent. Thus we *must not* have a happens-before dependency between the write to z (from node 1) and the read on z.

For an even more precarious example, consider Fig. 9c, which is a possible behaviour of LOCO's broadcast library. The final outcome $(a, b, c) = (1, 2, 1)$ is only possible if node 1 broadcasts $x = 1$ to node 2, and $x = 2$ to node 3 with a single broadcast. The broadcast is allowed to pick up the later value 2 since the CPU might run the command $x :=_{\text{SV}} 2$ before the NIC reads the value of x. As mentioned above, reading the result of a broadcast *must* create happens-before order so that we can preclude behaviours like in Fig. 9a. In this example, we thus need a sequence of dependencies: $x :=_{\text{SV}} 1 \rightarrow \texttt{Bcast}_{\text{SV}}(x) \rightarrow c :=_{\text{SV}} x \rightarrow y :=_{\text{SV}} c \rightarrow \texttt{Bcast}_{\text{SV}}(y) \rightarrow a :=_{\text{SV}} y \rightarrow x :=_{\text{SV}} 2 \rightarrow \texttt{Bcast}_{\text{SV}}(x) \rightarrow b :=_{\text{SV}} x$. This sequence seemingly contains a dependency cycle from $\texttt{Bcast}_{\text{SV}}(x)$ to itself, and thus any reasonable system of dependencies on events would not allow this valid behaviour.

We fix this apparent cycle by splitting the broadcast event into its four basic components called subevents: **(1)** reading x to send to node 2 (stamp aNLR_2); **(2)** writing x on node 2 (stamp aNRW_2); **(3)** reading x to send to node 3 (stamp aNLR_3); **(4)** writing x on node 3 (stamp aNRW_3). With this we can create a more fine-grain sequence of dependencies: $x :=_{\mathrm{sv}} 1 \longrightarrow \langle \mathtt{Bcast}_{\mathrm{sv}}(x), \mathsf{aNLR}_2 \rangle \longrightarrow \langle \mathtt{Bcast}_{\mathrm{sv}}(x), \mathsf{aNRW}_2 \rangle \longrightarrow c :=_{\mathrm{sv}} x \longrightarrow \ldots \longrightarrow x :=_{\mathrm{sv}} 2 \longrightarrow \langle \mathtt{Bcast}_{\mathrm{sv}}(x), \mathsf{aNLR}_3 \rangle \longrightarrow \langle \mathtt{Bcast}_{\mathrm{sv}}(x), \mathsf{aNRW}_3 \rangle \longrightarrow b :=_{\mathrm{sv}} x$. For each remote node the broadcast reads before writing, and we have a dependency between writing on node 2 and reading for node 3, but this does not create a dependency cycle at the level of the subevents and we can authorise the behaviour of Fig. 9c.

Stamps are shared by all libraries and also allow us to precisely define ppo, i.e. which pairs of effects are required to execute in order, even across libraries. For instance in example Fig. 9a we have a dependency $\langle \overline{z} := 1, \mathsf{aNRW}_2 \rangle \xrightarrow{\text{ppo}} \langle \mathtt{Bcast}_{\mathrm{sv}}(x), \mathsf{aNRW}_2 \rangle$ guaranteeing that the contents of z and x on node 2 are modified in order. However, note that this is more subtle than a dependency between events as the location x might still be read by the broadcast *before* the content of z is modified, i.e. $\langle \mathtt{Bcast}_{\mathrm{sv}}(x), \mathsf{aNLR}_2 \rangle \longrightarrow \langle \overline{z} := 1, \mathsf{aNRW}_2 \rangle \longrightarrow \langle \mathtt{Bcast}_{\mathrm{sv}}(x), \mathsf{aNRW}_2 \rangle$, as is allowed by the semantics of RDMA.

Modularity. A final challenge in developing Mowgli is to support modularity through both horizontal composition (the use of libraries in a client program) and vertical composition (the development of libraries using other libraries as a subcomponent). Mowgli presents a generic framework that is independent of a memory model to support such proofs through a locality theorem. It allows the simultaneous use of multiple libraries within a single program, and defines a semantics when the specification of a library is used in place of an implementation. Finally, it provides local methods for proving that a library implementation satisfies its specification.

3 The Mowgli Framework and the Shared Variable Library

In this section we define Mowgli's meta-language and general theory for modelling weak memory libraries, as well as its notion of compositionality that enables modular proofs. We note that our language and theory is generic and could be applied to other memory models. We present the syntax and semantics of Mowgli in §3.1 and model for formalising libraries in §3.2. Throughout the section, we use the shared variable library (sv) as a running example and define its consistency in §3.4. Then we present library abstraction in §3.5 and our main locality result in §3.6.

3.1 Syntax and Semantics

In this section, we present the syntax and semantics of our basic programming language. Our language is inspired by Cminor [Appel and Blazy 2007] and Yacovet [Stefanesco et al. 2024].

Programs. We assume a type Val of values, a type $\mathsf{Loc} \subseteq \mathsf{Val}$ of locations[2], and a type Method of methods. The syntax of sequential programs is given by the following grammar:

$$v, v_i \in \mathsf{Val} \qquad m \in \mathsf{Method} \qquad \mathsf{f} \in \mathsf{Val} \longrightarrow \mathsf{SeqProg} \qquad k \in \mathbb{N}^+$$
$$\mathsf{SeqProg} \ni \mathsf{p} ::= v \mid m(v_1, \ldots, v_k) \mid \mathtt{let}\ \mathsf{p}\ \mathsf{f} \mid \mathtt{loop}\ \mathsf{p} \mid \mathtt{break}_k\ v$$

A method call is parameterised by a sequence of input values. In later sections, we will instantiate m to basic operations such as read and write, as well as operations corresponding to method calls of a high-level library.

For a function f mapping values to sequential programs, the syntax `let p f` denotes the execution of p with an output that is then used as an input for f. This constructor is a generalisation of

[2]In Mowgli, every argument of a method call is a value. Thus identifiers ($x, y, \ldots$) are called "locations" by the libraries but are seen as values by the meta-language.

the more standard let-in syntax, and for a program $\mathtt{p}_2$ with a free meta-variable x we can define $\mathtt{let}\, x = \mathtt{p}_1\, \mathtt{in}\, \mathtt{p}_2$ as $\mathtt{let}\, \mathtt{p}_1\ (\lambda v.\mathtt{p}_2[x := v])$. We can also model sequential composition, i.e. $\mathtt{p}_1; \mathtt{p}_2$, as syntactic sugar for $\mathtt{let}\, \mathtt{p}_1\ (\lambda\_.\ \mathtt{p}_2)$ using a constant function that discards its input. The syntax $\mathtt{let}\, \mathtt{p}\, \mathtt{f}$ also allows programs to perform branching and pattern-matching, via a function mapping different kinds of values to different continuations. In particular, $\mathtt{if}\ v\ \mathtt{then}\ \mathtt{p}_1\ \mathtt{else}\ \mathtt{p}_2$ can be taken as syntactic sugar for $\mathtt{let}\, v\ \{\mathtt{true} \mapsto \mathtt{p}_1, \mathtt{false} \mapsto \mathtt{p}_2\}$.

Finally, our syntax includes $\mathtt{loop}\, \mathtt{p}$ that infinitely executes the program p, as well as the $\mathtt{break}_k\ v$ construct which exits k levels of nested loops and returns v. While uncommon, these constructs can be used to define usual `while` and `for` loops.

We assume top-level concurrency. We assume a fixed number T of threads and let $\mathsf{Tid} \triangleq \{1, 2, \ldots, T\}$ be the set of all threads. A concurrent program is thus given by a tuple $\widetilde{\mathtt{p}} = \langle \mathtt{p}_1, \ldots, \mathtt{p}_T \rangle$, where each thread t corresponds to a program $\mathtt{p}_t \in \mathsf{SeqProg}$. Note that we allow libraries to discriminate threads, and so the position of a program in $\widetilde{\mathtt{p}}$ matters, e.g. the program $\langle \mathtt{p}_1, \ldots, \mathtt{p}_T \rangle$ is *not* equivalent to $\langle \mathtt{p}_T, \ldots, \mathtt{p}_1 \rangle$. For instance, a pair of RDMA threads have different interactions depending on whether they run on the same node or not.

Example 3.1 (Shared Variables). For our RDMA libraries, we assume a set of nodes, Node, of fixed size. Each thread t is associated to a node $\mathtt{n}(t)$. The sv library uses the following methods:

$$m(\widetilde{v}) ::= \mathtt{Write}_{\mathrm{sv}}(x, v) \mid \mathtt{Read}_{\mathrm{sv}}(x) \mid \mathtt{Bcast}_{\mathrm{sv}}(x, d, \{n_1, \ldots, n_k\}) \mid \mathtt{Wait}_{\mathrm{sv}}(d) \mid \mathtt{GF}_{\mathrm{sv}}(\{n_1, \ldots, n_k\})$$

$\mathtt{Write}_{\mathrm{sv}}(x, v)$ writes a new value v to the location x of the current node. $\mathtt{Read}_{\mathrm{sv}}(x)$ reads the location x of the current node and returns its value. $\mathtt{Bcast}_{\mathrm{sv}}(x, d, \{n_1, \ldots, n_k\})$ broadcasts the local value of x and overwrites the values of the copies of x on the nodes $\{n_1, \ldots, n_k\}$, which might include the local node. $\mathtt{Wait}_{\mathrm{sv}}(d)$ waits for previous broadcasts of the thread marked with the same work identifier $d \in \mathsf{Wid}$. As mentioned in the overview, this operation only guarantees that the local values of the broadcasts have been read, but not that remote copies have been modified. Finally, the global fence operation $\mathtt{GF}_{\mathrm{sv}}(\{n_1, \ldots, n_k\})$ ensures every previous operation of the thread towards one of the nodes in the argument is fully finished, including the writing part of broadcasts.

Plain Executions. The semantics of a program is given by an execution, which is a graph over events. Each event has a label taken from the set $\mathsf{Lab} \triangleq \mathsf{Method} \times \mathsf{Val}^* \times \mathsf{Val}$, i.e. a triple comprising the method, the input values, and the output value. Labels are used to define events, which are elements of the set $\mathsf{Event} \triangleq \mathsf{Tid} \times \mathsf{EventId} \times \mathsf{Lab}$, where $\mathsf{EventId} \triangleq \mathbb{N}$. For each event $\langle t, \iota, l \rangle \in \mathsf{Event}$, we have that $t \in \mathsf{Tid}$ is the thread that executes the label $l \in \mathsf{Lab}$, and ι is a unique identifier for the event. For an event $\mathtt{e} = \langle t, \iota, l \rangle$, we note $\mathtt{t}(\mathtt{e}) \triangleq t$.

Definition 3.2. We say that $\langle E, \mathrm{po} \rangle$ is a *plain execution* iff $E \subseteq \mathsf{Event}$, $\mathrm{po} \subseteq E \times E$, and $\mathrm{po} = \bigcup_{t \in \mathsf{Tid}} \mathrm{po}|_t$ where every $\mathrm{po}|_t$ (i.e. po restricted to the events of thread t) is a total order.

Here, po represents *program order* i.e. $\langle \mathtt{e}_1, \mathtt{e}_2 \rangle \in \mathrm{po}$ iff $\mathtt{e}_1$ is executed before $\mathtt{e}_2$ by the same thread.

We write $\emptyset_G \triangleq \langle \emptyset, \emptyset \rangle$ for the empty execution and $\{\mathtt{e}\}_G \triangleq \langle \{\mathtt{e}\}, \emptyset \rangle$ for the execution with a single event e. Given two executions, $G_1 = \langle E_1, \mathrm{po}_1 \rangle$ and $G_2 = \langle E_2, \mathrm{po}_2 \rangle$, with disjoint sets of events (i.e. $E_1 \cap E_2 = \emptyset$), we define their sequential composition, $G_1; G_2$, by ordering all events of G_1 before those of G_2. Similarly, we define their parallel composition, $G_1 \| G_2$, by taking the union of G_1 and G_2. That is,

$$G_1; G_2 \triangleq \langle E_1 \cup E_2, \mathrm{po}_1 \cup \mathrm{po}_2 \cup (E_1 \times E_2) \rangle \qquad G_1 \| G_2 \triangleq \langle E_1 \cup E_2, \mathrm{po}_1 \cup \mathrm{po}_2 \rangle$$

The plain semantics of a program p executed by a thread t is given by $[\![\mathtt{p}]\!]_t$, which is a set of pairs of the form $\langle r, G \rangle$, where r is the output and G is a plain execution. This set represents all conceivable unfoldings of the program into method calls, even those that will be rejected by the semantics of

the corresponding libraries. Each output is a pair $\langle v, k\rangle$, where v is a value and k a break number, indicating the program terminates by requesting to exit k nested loops and returning the value v.

$$\begin{aligned}
[\![v]\!]_t &\triangleq \{\langle\langle v, 0\rangle, \emptyset_G\rangle\} \qquad [\![\mathtt{break}_k\ v]\!]_t \triangleq \{\langle\langle v, k\rangle, \emptyset_G\rangle\} \\
[\![m(\widetilde{v})]\!]_t &\triangleq \{\langle\langle v', 0\rangle, \{\langle t, \iota, \langle m, \widetilde{v}, v'\rangle\rangle\}_G\rangle \mid v' \in \mathsf{Val} \ \wedge\ \iota \in \mathsf{EventId}\} \\
[\![\mathtt{let\ p\ f}]\!]_t &\triangleq \left\{\langle r, G_1; G_2\rangle \;\middle|\; \langle\langle v, 0\rangle, G_1\rangle \in [\![\mathtt{p}]\!]_t \ \wedge\ \langle r, G_2\rangle \in [\![\mathtt{f}\ v]\!]_t\right\} \\
&\quad \cup \left\{\langle\langle v, k\rangle, G_1\rangle \;\middle|\; \langle\langle v, k\rangle, G_1\rangle \in [\![\mathtt{p}]\!]_t \ \wedge\ k \neq 0\right\} \\
[\![\mathtt{loop\ p}]\!]_t &\triangleq \bigcup_{j\in\mathbb{N}} \left\{\langle\langle v, k\rangle, G_0; \ldots; G_j\rangle \;\middle|\; (\forall 0 \leq i < j.\ \langle\langle _, 0\rangle, G_i\rangle \in [\![\mathtt{p}]\!]_t) \ \wedge\ \langle\langle v, k+1\rangle, G_j\rangle \in [\![\mathtt{p}]\!]_t\right\}
\end{aligned}$$

The execution of a value v simply returns $\langle v, 0\rangle$ with an empty graph. Similarly, the execution of $\mathtt{break}_k\ v$ returns $\langle v, k\rangle$ with a non-zero break number and an empty graph.

The plain semantics of $[\![m(\widetilde{v})]\!]_t$ considers every value v' as a possible output of the method call. For each, we can create a graph G with a single event $\langle t, \_, \langle m, \widetilde{v}, v'\rangle\rangle$, and the corresponding output for the program is then $\langle v', 0\rangle$ with a break number of 0.

The execution of `let` p f has two kinds of plain semantics. Either the execution of p requests a break, i.e. $\langle\langle v, k\rangle, G_1\rangle \in [\![\mathtt{p}]\!]_t$ with $k \neq 0$, in which case `let` p f breaks as well with the same output. Or p terminates with a break number of zero, and the output value v of p is given to f. In this second case, the plain execution of `let` p f is the sequential composition of the plain executions for p and (f v), and its output value is the one of (f v).

Finally, the execution of `loop` p can be unfolded and corresponds to the execution of p any number $j + 1$ of times. The first j times, p returns without requesting a break and its output value is ignored. The $(j + 1)^{\text{th}}$ execution of p returns a value v and break number $k + 1$, and `loop` p propagates $\langle v, k\rangle$ with a decremented break number. The plain execution of the loop is then the sequential composition of the plain executions of the $j + 1$ iterations of p.

We lift the plain semantics to the level of concurrent programs and define

$$[\![\widetilde{\mathtt{p}}]\!] \triangleq \left\{\langle\langle v_1, \ldots, v_T\rangle, \|_{t\in\mathsf{Tid}}\ G_t\rangle \;\middle|\; \forall t \in \mathsf{Tid}.\langle\langle v_t, 0\rangle, G_t\rangle \in [\![\mathtt{p}_t]\!]_t\right\}$$

Concurrent programs only properly terminate if each thread terminates with a break number of 0. In which case, the output of the concurrent program is the parallel composition of the values and plain executions of the different threads.

Executions. We generate executions from plain executions by (1) extending the model with subevents, then (2) introducing additional relations describing synchronisation and happens-before order. We will later define consistency conditions for executions in the context of libraries.

We assume a fixed set of stamps, $\mathsf{Stamp} = \{a_1, \ldots\}$, and a relation $\mathsf{to} \subseteq \mathsf{Stamp} \times \mathsf{Stamp}$. We will use stamps to define subevents and to to define preserved program order over subevents within an execution.

Definition 3.3. We say that $\langle E, \mathsf{po}, \mathtt{stmp}, \mathsf{so}, \mathsf{hb}\rangle$ is an *execution* iff each of the following holds:

- $\langle E, \mathsf{po}\rangle$ is a plain execution.
- $\mathtt{stmp} : E \to \mathcal{P}(\mathsf{Stamp})$ is a function that associates each event with a non-empty set of stamps and induces a set of *subevents*, $\mathsf{SEvent} \triangleq \{\langle \mathtt{e}, a\rangle \mid \mathtt{e} \in E \wedge a \in \mathtt{stmp}(\mathtt{e})\}$.
- $\mathsf{so} \subseteq \mathsf{SEvent} \times \mathsf{SEvent}$ and $\mathsf{hb} \subseteq \mathsf{SEvent} \times \mathsf{SEvent}$ are relations on SEvent defining *synchronisation order* and *happens-before order*, respectively.

To define consistency, we must ultimately relate po, so, and hb. However, in many weak memory models such as RDMA, including all of po into hb is too restrictive. We therefore make use of a

Second Stamp

First Stamp		to	single					families					
			1	2	3	4	5	6	7	8	9	10	11
			aCR	aCW	aCAS	aMF	aWT	aNLR_n	aNRW_n	aNRR_n	aNLW_n	aRF_n	aGF_n
single	A	aCR	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	B	aCW	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
	C	aCAS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	D	aMF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	E	aWT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
families	F	aNLR_n	✗	✗	✗	✗	✗	SN	SN	SN	SN	SN	SN
	G	aNRW_n	✗	✗	✗	✗	✗	✗	SN	SN	SN	✗	SN
	H	aNRR_n	✗	✗	✗	✗	✗	✗	✗	✗	SN	SN	SN
	I	aNLW_n	✗	✗	✗	✗	✗	✗	✗	✗	SN	✗	SN
	J	aRF_n	✗	✗	✗	✗	✗	SN	SN	SN	SN	SN	SN
	K	aGF_n	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Fig. 10. Stamp order to for the RDMA libraries. Lines indicate the earlier stamp, columns the later. A cell marked ✓ indicates that the stamps are ordered, and that the po ordering of subevents with these stamps is preserved. A cell marked ✗ indicates that the stamps are not ordered, and that such subevents can execute out of order. Finally, SN indicates the stamps are ordered iff they have the same node index.

weaker relation called *preserved program order*, ppo $\subseteq$ SEvent $\times$ SEvent, which we derive from po and to as follows:

$$\mathsf{ppo} \triangleq \{\langle\langle \mathsf{e}_1, a_1\rangle, \langle \mathsf{e}_2, a_2\rangle\rangle \mid \langle \mathsf{e}_1, \mathsf{e}_2\rangle \in \mathsf{po} \wedge a_1 \in \mathtt{stmp}(\mathsf{e}_1) \wedge a_2 \in \mathtt{stmp}(\mathsf{e}_2) \wedge \langle a_1, a_2\rangle \in \mathsf{to}\}$$

For our RDMA libraries, we define 11 kinds of stamps. We have aCR representing a CPU read; aCW representing a CPU write; aCAS for an atomic read-modify-write operation; aMF for a TSO memory fence; aWT for a `wait` operation; aNLR_n for a NIC local read; aNRW_n for a NIC remote write; aNRR_n for a NIC remote read; aNLW_n for a NIC local write; aRF_n for a NIC remote fence; and aGF_n for a global fence operation. The last 6 are families of stamps, as we create a different copy for each node $n \in$ Node.

The stamp order to we use is defined in Fig. 10. We note ✓ when two stamps are ordered, ✗ when they are not ordered, and SN when they are ordered iff they have the same index node. For instance, the ✗ in cell B1 indicates that when a CPU write is in program order before a CPU read, there is no ordering guarantee between the two operations, as we assume the CPUs follow the TSO memory model, and the read might execute first.

Example 3.4 (ppo for Shared Variables). For the sv library, we use the stamping function $\mathtt{stmp}_{\mathsf{SV}}$:

$$\begin{aligned}
\mathtt{stmp}_{\mathsf{SV}}(\langle _, _, \langle \mathtt{Write}_{\mathsf{sv}}, _, _\rangle\rangle) &= \{\mathsf{aCW}\} \\
\mathtt{stmp}_{\mathsf{SV}}(\langle _, _, \langle \mathtt{Read}_{\mathsf{sv}}, _, _\rangle\rangle) &= \{\mathsf{aCR}\} \\
\mathtt{stmp}_{\mathsf{SV}}(\langle _, _, \langle \mathtt{Wait}_{\mathsf{sv}}, _, _\rangle\rangle) &= \{\mathsf{aWT}\} \\
\mathtt{stmp}_{\mathsf{SV}}(\langle _, _, \langle \mathsf{GF}_{\mathsf{sv}}, (\{n_1, \ldots, n_k\}), _\rangle\rangle) &= \{\mathsf{aGF}_{n_1}, \ldots, \mathsf{aGF}_{n_k}\} \\
\mathtt{stmp}_{\mathsf{SV}}(\langle _, _, \langle \mathtt{Bcast}_{\mathsf{sv}}, (_, _, \{n_1, \ldots, n_k\}), _\rangle\rangle) &= \{\mathsf{aNLR}_{n_1}, \mathsf{aNRW}_{n_1}, \ldots, \mathsf{aNLR}_{n_k}, \mathsf{aNRW}_{n_k}\}
\end{aligned}$$

Broadcasts are associated with a NIC local read and NIC remote write stamp for each remote node they are broadcasting towards. Similarly, global fence operations are associated with a global fence stamp for each node.

With this, the stamp order is enough to enforce the behaviour of the global fence. If we have a program $\mathtt{Bcast}_{\mathsf{sv}}(x, d, \{\ldots, n, \ldots\}); \mathsf{GF}_{\mathsf{sv}}(\{\ldots, n, \ldots\})$, the plain execution has two events e_{BR} and e_{GF}, and the definitions of $\mathtt{stmp}_{\mathsf{SV}}$ and to (cell G11 in Fig. 10) imply $\langle \mathsf{e}_{BR}, \mathsf{aNRW}_n\rangle \xrightarrow{\mathsf{ppo}} \langle \mathsf{e}_{GF}, \mathsf{aGF}_n\rangle$.

3.2 Libraries

In this section, we describe how libraries and library consistency are modelled in our framework.

Definition 3.5. We say that a triple $\langle M, \mathtt{loc}, C\rangle$ is a *library* iff each of the following holds.

(1) $M \subseteq \mathsf{Method}$ is a set of *methods*.
(2) $\mathtt{loc} : \mathsf{Event}|_M \to \mathcal{P}(\mathsf{Loc})$ is a function associating each method call to a set of locations accessed by the method call.
(3) C is a *consistency predicate* over executions, respecting the following two properties.
   - Monotonicity: If $\langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle \in C$ (i.e. is consistent), and $(\mathrm{ppo} \cup \mathrm{so})^+ \subseteq \mathrm{hb}' \subseteq \mathrm{hb}$, then $\langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}'\rangle \in C$.
   - Decomposability: If $\langle (E_1 \uplus E_2), \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle \in C$ and $\mathtt{loc}(E_1) \cap \mathtt{loc}(E_2) = \emptyset$, then $\langle E_1, \mathrm{po}|_{E_1}, \mathtt{stmp}|_{E_1}, \mathrm{so}|_{E_1}, \mathrm{hb}|_{E_1}\rangle \in C$.

Usually, including for all of the examples considered in this paper, the locations accessed by a method call are a subset of its arguments. E.g., we say that $\mathtt{Write}(x, v)$ only accesses x. Monotonicity states that removing constraints cannot disallow a behaviour; this is trivially respected by all reasonable libraries. Decomposability states that method calls manipulating different locations can be considered independently. Crucially, this means combining independent programs *cannot* create additional behaviours; a prerequisite for modular verification. This holds for almost all libraries, and usually only breaks when programs have access to meta-information (e.g. the number of instructions of the whole program).

However, decomposability does *not* hold for RDMA$^{\mathrm{TSO}}$. As show in Fig. 2, the program $\overline{z} := x; \mathtt{Poll}(2); x := 1$ does not allow the outcome $z = 1$, while a combined program $\mathsf{p}; \overline{z} := x; \mathtt{Poll}(2); x := 1$ might, even when p seems independent (i.e. does not use locations z and x). This composition problem fundamentally prevents modular verification of RDMA$^{\mathrm{TSO}}$ programs. It is the reason we develop the alternative semantics of RDMA$^{\mathrm{WAIT}}$, while ensuring the two semantics are as close as possible.

Notation. For a library L, we have $\mathsf{Event}|_{L.M} = \{\langle \_, \_, \langle m, \_, \_\rangle\rangle \in \mathsf{Event} \mid m \in L.M\}$. We use $\mathsf{Event}|_L$ to refer to $\mathsf{Event}|_{L.M}$. Moreover, $\mathtt{loc}(\mathsf{e})$ is used to denote $L.\mathtt{loc}(\mathsf{e})$, where L is the library containing e (i.e. $\mathsf{e} \in \mathsf{Event}|_L$) and for $E \subseteq \mathsf{Event}$, we define $\mathtt{loc}(E) \triangleq \bigcup_{\mathsf{e}\in E} \mathtt{loc}(\mathsf{e})$. From this, we can also define the locations $\mathtt{loc}(\widetilde{\mathsf{p}})$ of a program $\widetilde{\mathsf{p}}$ as $\mathtt{loc}(\widetilde{\mathsf{p}}) \triangleq \bigcup_{\langle -, \langle E, - \rangle\rangle \in [\![\widetilde{\mathsf{p}}]\!]} \mathtt{loc}(E)$.

Given a relation r and a set A, we write r^+ for the transitive closure of r; r^* for its reflexive transitive closure; r^{-1} for the inverse of r; $r|_A$ for $r \cap (A \times A)$; and $[A]$ for the identity relation on A, i.e. $\{\langle a, a\rangle \mid a \in A\}$. We write $r_1; r_2$ for the relational composition of r_1 and r_2: $\{\langle a, b\rangle \mid \exists c.\, \langle a, c\rangle \in r_1 \wedge \langle c, b\rangle \in r_2\}$.

Consistent Execution. Two libraries are *compatible* if their sets of methods are disjoint. We use Λ to denote a set of pairwise compatible libraries.

Definition 3.6. Let Λ be a set of pairwise compatible libraries. An execution $\langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle$ is Λ-*consistent* iff each of the following holds.

- $(\mathrm{ppo} \cup \mathrm{so})^+ \subseteq \mathrm{hb}$ and hb is a strict partial order (i.e. both irreflexive and transitive).
- $E = \bigcup_{L\in\Lambda} E|_L$ and $\mathrm{so} = \bigcup_{L\in\Lambda} \mathrm{so}|_L$.
- For all $L \in \Lambda$, we have $\langle E|_L, \mathrm{po}|_L, \mathtt{stmp}|_L, \mathrm{so}|_L, \mathrm{hb}|_L\rangle \in L.C$.

Although the definition of Λ-consistency allows hb relations that are bigger than $(\mathrm{ppo} \cup \mathrm{so})^+$, we usually have $\mathrm{hb} = (\mathrm{ppo} \cup \mathrm{so})^+$ for the program executions we are interested in.

Given a concurrent program $\widetilde{\mathsf{p}}$ using libraries Λ, we note $\mathtt{outcome}_\Lambda(\widetilde{\mathsf{p}})$ the set of all output values of its Λ-consistent executions.

$$\mathtt{outcome}_\Lambda(\widetilde{\mathsf{p}}) \triangleq \left\{\widetilde{v} \;\middle|\; \exists \langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle\ \Lambda\text{-consistent}.\ \langle \widetilde{v}, \langle E, \mathrm{po}\rangle\rangle \in [\![\widetilde{\mathsf{p}}]\!]\right\}$$

3.3 The RDMA$^{\text{WAIT}}$ Library

RDMA$^{\text{WAIT}}$ is used as the lowest library of our tower of abstraction (Fig. 1). As mentioned in §3.4, it is the implementation target for the shared variable library (SV). It is an adaptation of RDMA$^{\text{TSO}}$ where the `poll` instruction is replaced by a more intuitive `wait` operation.

The RDMA$^{\text{WAIT}}$ library uses the following 8 methods.

$$\begin{aligned} m(\widetilde{v}) ::= {} & \mathtt{Write}(x, v) \mid \mathtt{Read}(x) \mid \mathtt{CAS}(x, v_1, v_2) \mid \mathtt{Mfence}() \\ & \mid \mathtt{Get}(x, y, d) \mid \mathtt{Put}(x, y, d) \mid \mathtt{Wait}(d) \mid \mathtt{Rfence}(n) \end{aligned}$$

The first line covers usual TSO operations: $\mathtt{Write}(x, v)$ is a CPU write; $\mathtt{Read}(x)$ is a CPU read; $\mathtt{CAS}(x, v_1, v_2)$ is an atomic compare-and-swap operation that overwrites x to v_2 iff x contained v_1, and returns the old value of x; and $\mathtt{Mfence}()$ is a TSO memory fence flushing the store buffer.

The second line covers RDMA-specific operations: $\mathtt{Get}(x, y, d)$ (noted $x :=^d \overline{y}$ in our examples) is a get[3] operation with work identifier d performing a NIC remote read on y and a NIC local write on x; similarly $\mathtt{Put}(x, y, d)$ (noted $\overline{x} :=^d y$) is a put operation with work identifier d performing a NIC local read on y and a NIC remote write on x; $\mathtt{Wait}(d)$ waits for previous operations with work identifier d; and finally $\mathtt{Rfence}(n)$ is an RDMA remote fence for the communication channel towards n that does not block the CPU.

We assume that each location x is associated with a specific node $\mathtt{n}(x)$. From this, given $\langle E, \mathrm{po}\rangle$, there is a single valid stamping function $\mathtt{stmp}_{\mathsf{RL}}$. Notably we have $\mathtt{stmp}_{\mathsf{RL}}(\mathtt{Get}(x, y, d)) = \left\{\mathtt{aNRR}_{\mathtt{n}(y)}, \mathtt{aNLW}_{\mathtt{n}(y)}\right\}$ and $\mathtt{stmp}_{\mathsf{RL}}(\mathtt{Put}(x, y, d)) = \left\{\mathtt{aNLR}_{\mathtt{n}(x)}, \mathtt{aNRW}_{\mathtt{n}(x)}\right\}$. Put and get operations perform both a NIC read and a NIC write, and as such are associated to two stamps, where the remote node can be deduced from the location. Also, a succeeding `CAS` has a single stamp `aCAS`, while a failing `CAS` has stamps $\{\mathtt{aMF}, \mathtt{aCR}\}$, as it behaves as both a memory fence (`aMF`) and a CPU read (`aCR`).

The formal semantics requires several functions and relations: v_R, v_W, rf, and mo, with roles similar to the semantics of SV (*cf.* §3.4), as well as the *NIC-flush-order* relation nfo representing the PCIe guarantees that NIC reads flush previous NIC writes. The consistency predicate for RDMA$^{\text{WAIT}}$ is then stated from these relations and some derived relations, similarly to §3.4.

3.4 Example: Consistency for Shared Variables

As mentioned in Example 3.1, SV uses the methods $M = \{\mathtt{Write}_{\mathrm{sv}}, \mathtt{Read}_{\mathrm{sv}}, \mathtt{Bcast}_{\mathrm{sv}}, \mathtt{Wait}_{\mathrm{sv}}, \mathtt{GF}_{\mathrm{sv}}\}$. Since only the method and arguments matter for the location function, we use $\mathtt{loc}(m(\widetilde{v}))$ to denote $\mathtt{loc}(\langle \_, \_, \langle m, \widetilde{v}, \_\rangle\rangle)$, where $\mathtt{loc}(\mathtt{Write}_{\mathrm{sv}}(x, \_)) = \mathtt{loc}(\mathtt{Read}_{\mathrm{sv}}(x)) = \mathtt{loc}(\mathtt{Bcast}_{\mathrm{sv}}(x, \_, \_)) = \{x\}$ for events accessing a location x, and $\mathtt{loc}(\mathtt{e}) = \emptyset$ otherwise for methods $\mathtt{Wait}_{\mathrm{sv}}$ and $\mathtt{GF}_{\mathrm{sv}}$.

Notation. For a subevent s, we note s.e and s.a its two components. Given an execution $\mathcal{G} = \langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle$ and a stamp a, we write $\mathcal{G}.a$ for $\{\mathtt{s} \in \mathcal{G}.\mathtt{SEvent} \mid \mathtt{s}.a = a\}$. For families, by abuse of notation, we also write e.g. $\mathcal{G}.\mathtt{aNRR}$ for $\bigcup_{n \in \mathtt{Node}} \mathcal{G}.\mathtt{aNRR}_n$. We extend the notation `loc` to subevents by writing $\mathtt{loc}(\mathtt{s})$ for $\mathtt{loc}(\mathtt{s.e})$. We define the set of *reads* as $\mathcal{G}.\mathcal{R} \triangleq \mathcal{G}.\mathtt{aCR} \cup \mathcal{G}.\mathtt{aCAS} \cup \mathcal{G}.\mathtt{aNLR} \cup \mathcal{G}.\mathtt{aNRR}$ and *writes* as $\mathcal{G}.\mathcal{W} \triangleq \mathcal{G}.\mathtt{aCW} \cup \mathcal{G}.\mathtt{aCAS} \cup \mathcal{G}.\mathtt{aNLW} \cup \mathcal{G}.\mathtt{aNRW}$. We write $\mathcal{G}.\mathcal{W}_x \triangleq \{\mathtt{s} \in \mathcal{G}.\mathcal{W} \mid \mathtt{loc}(\mathtt{s}) = \{x\}\}$ to constrain the set to writes on a specific location x. We also use

[3]In the RDMA specification, `Get` and `Put` are referred to as respectively "RDMA Read" and "RDMA Write" operations. We use the terms get and put to prevent confusion, as each of these perform both a read and a write subevents.

$|_t$ to restrict a set or relation to a specific thread. E.g. $E|_t = \{\mathsf{e} \mid \mathsf{e} \in E \wedge \mathtt{t}(\mathsf{e}) = t\}$ and $\mathrm{po}|_t = [E|_t]; \mathrm{po}; [E|_t]$.

For the sv library, we additionally define $\mathcal{G}.\mathcal{W}^n \triangleq \{\langle \mathsf{e}, \mathtt{aCW}\rangle \mid \mathtt{n}(\mathtt{t}(\mathsf{e})) = n\} \cup \mathcal{G}.\mathtt{aNRW}_n$ as the set of write subevents occurring on node n. This includes CPU writes on the node, as well as broadcast writes towards n from all threads. We also note $\mathcal{G}.\mathcal{W}^n_x \triangleq \mathcal{G}.\mathcal{W}_x \cap \mathcal{G}.\mathcal{W}^n$ as expected. Similarly, $\mathcal{G}.\mathcal{R}^n \triangleq \{\mathsf{s} \mid \mathsf{s} \in \mathcal{G}.\mathcal{R} \wedge \mathtt{n}(\mathtt{t}(\mathsf{s})) = n\}$ covers reads occurring on n, either by a CPU read or as part of a broadcast.

Consistency. We now work towards a definition of consistency for shared variables.

Definition 3.7. For an execution $\mathcal{G} = \langle E, \mathrm{po}, \mathtt{stmp}_{\mathsf{SV}}, \_, \_\rangle$, we define the following:

- The *value-read* function $\mathsf{v}_\mathsf{R} : \mathcal{G}.\mathcal{R} \to \mathrm{Val}$ that associates each read subevent with the value returned, if available, i.e. if $\mathsf{e} = \langle \_, \_, \langle \mathtt{Read}_{\mathsf{sv}}, \_, v\rangle\rangle$, then $\mathsf{v}_\mathsf{R}(\mathsf{e}) = v$.
- The *value-written* function $\mathsf{v}_\mathsf{W} : \mathcal{G}.\mathcal{W} \to \mathrm{Val}$ that associates each write subevent with a value $\mathcal{G}$, i.e. if $\mathsf{e} = \langle \_, \_, \langle \mathtt{Write}_{\mathsf{sv}}, (\_, v), \_\rangle\rangle$, then $\mathsf{v}_\mathsf{W}(\mathsf{e}) = v$.
- A *reads-from* relation, $\mathrm{rf} \triangleq \bigcup_n \mathrm{rf}^n$, where each $\mathrm{rf}^n \subseteq \mathcal{G}.\mathcal{W}^n \times \mathcal{G}.\mathcal{R}^n$ is a relation on subevents of the same location and node with matching values, i.e. if $\langle \mathsf{s}_1, \mathsf{s}_2\rangle \in \mathrm{rf}^n$ then $\mathtt{loc}(\mathsf{s}_1) = \mathtt{loc}(\mathsf{s}_2)$ and $\mathsf{v}_\mathsf{W}(\mathsf{s}_1) = \mathsf{v}_\mathsf{R}(\mathsf{s}_2)$.
- A *modification-order* relation $\mathrm{mo} \triangleq \bigcup_{x,n} \mathrm{mo}^n_x$ describing the order in which writes on x on node n reach memory.

We define *well-formedness* for rf and mo as follows. For each remote, a broadcast writes the corresponding read value: if $\mathsf{s}_1 = \langle \mathsf{e}, \mathtt{aNLR}_n\rangle \in \mathcal{G}.\mathrm{SEvent}$ and $\mathsf{s}_2 = \langle \mathsf{e}, \mathtt{aNRW}_n\rangle \in \mathcal{G}.\mathrm{SEvent}$, then $\mathsf{v}_\mathsf{R}(\mathsf{s}_1) = \mathsf{v}_\mathsf{W}(\mathsf{s}_2)$. Each rf^n is functional on its range, i.e. every read in $\mathcal{G}.\mathcal{R}^n$ is related to at most one write in $\mathcal{G}.\mathcal{W}^n$. If a read is not related to a write, it reads the initial value of zero, i.e. if $\mathsf{s}_2 \in \mathcal{G}.\mathcal{R}^n \wedge \langle \_, \mathsf{s}_2\rangle \notin \mathrm{rf}^n$ then $\mathsf{v}_\mathsf{R}(\mathsf{s}_2) = 0$. Finally, each mo^n_x is a strict total order on $\mathcal{G}.\mathcal{W}^n_x$.

We further define the *reads-from-internal* relation as $\mathrm{rf_i} \triangleq [\mathtt{aCW}]; (\mathrm{po} \cap \mathrm{rf}); [\mathtt{aCR}]$ (which corresponds to CPU reads and writes using the same TSO store buffer), and the *reads-from-external* relation as $\mathrm{rf_e} \triangleq \mathrm{rf} \setminus \mathrm{rf_i}$. As we shall see in Def. 3.8, $\mathrm{rf_i}$ does *not* contribute to synchronisation order, whereas $\mathrm{rf_e}$ does. Moreover, given an execution $\mathcal{G}$ and well-formed rf and mo, we derive additional relations.

$$\mathrm{pf} \triangleq \left\{ \langle\langle \mathsf{e}_1, \mathtt{aNLR}_n\rangle, \langle \mathsf{e}_2, \mathtt{aWT}\rangle\rangle \;\middle|\; \langle \mathsf{e}_1, \mathsf{e}_2\rangle \in \mathrm{po} \wedge \begin{pmatrix} \exists d.\ \mathsf{e}_1 = \langle \_, \_, \langle \mathtt{Bcast}_{\mathsf{sv}}, (\_, \_, d), \_\rangle\rangle \\ \wedge\ \mathsf{e}_2 = \langle \_, \_, \langle \mathtt{Wait}_{\mathsf{sv}}, (d), \_\rangle\rangle \end{pmatrix} \right\}$$

$$\mathrm{rb}^n \triangleq \left\{ \langle r, w\rangle \in \mathcal{G}.\mathcal{R}^n \times \mathcal{G}.\mathcal{W}^n \;\middle|\; \begin{array}{c} \mathtt{loc}(r) = \mathtt{loc}(w) \\ \wedge\ \left(\langle r, w\rangle \in ((\mathrm{rf}^n)^{-1}; \mathrm{mo}^n) \vee r \notin \mathtt{img}(\mathrm{rf}^n)\right) \end{array} \right\} \qquad \mathrm{rb} \triangleq \bigcup_n \mathrm{rb}^n$$

$$\mathrm{iso} \triangleq \{\langle\langle \mathsf{e}, \mathtt{aNLR}_n\rangle, \langle \mathsf{e}, \mathtt{aNRW}_n\rangle\rangle \mid \mathsf{e} = \langle \_, \_, \langle \mathtt{Bcast}_{\mathsf{sv}}, (\_, \_, \{\ldots, n, \ldots\}), \_\rangle\rangle \in E\}$$

The *polls-from* relation pf states that a $\mathtt{Wait}_{\mathsf{sv}}$ operation synchronises with the NIC local read subevents of previous broadcasts that use the same work identifier. The *reads-before* relation rb states that a read r executes before a specific write w on the same node and location. This is either because r reads the initial value of 0, or because r reads from a write that is mo-before w. Finally, the *internal-synchronisation-order* relation iso states that, within a broadcast, for each remote node the reading part occurs before the writing part.

We can then define the consistency predicate $\mathrm{sv}.\mathcal{C}$ as follows.

Definition 3.8 (sv-consistency). $\langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle$ is sv-consistent if:

- $\mathtt{stmp} = \mathtt{stmp}_{\mathsf{SV}}$ (defined in §3.1);
- there exists well-formed v_R, v_W, rf, and mo, such that $[\mathtt{aCR}]; (\mathrm{po}^{-1} \cap \mathrm{rb}); [\mathtt{aCW}] = \emptyset$ and $\mathrm{so} = \mathrm{iso} \cup \mathrm{rf_e} \cup \mathrm{pf} \cup \mathrm{rb} \cup \mathrm{mo}$.

It is straightforward to check that this consistency predicate satisfies monotonicity and decomposability. For CPU reads and writes, we ask that rb does not contradict the program order. E.g., a program $\mathtt{Write}_{\mathsf{sv}}(x,1);\mathtt{Read}_{\mathsf{sv}}(x)$ must return 1 and cannot return 0, even if the semantics of TSO allows for the read to finish before the write.

There is no need to explicitly include conditions on hb in the consistency of the library, as the global consistency condition (*cf.* Def. 3.6) already enforces that $(\mathsf{ppo} \cup \mathsf{so} \cup \mathsf{hb})^+$ is irreflexive.

3.5 Library Implementations

We now describe a mechanism for implementing the method calls of a library by an implementation. Our ideas build on Yacovet [Stefanesco et al. 2024], but have been adapted to our setting, which comprises a much weaker happens-before relation (based on ppo instead of po). In particular, Mowgli's notions of implementation, soundness, and abstraction are similar to Yacovet (but simpler), but the notion of "local soundness" is more complicated due to the use of ppo and subevents.

An implementation for a library L is a function $I : (\mathsf{Tid} \times L.M \times \mathsf{Val}^*) \to \mathsf{SeqProg}$ associating every method call of the library L to a sequential program.

Definition 3.9. We say that I is *well defined* for a library L using Λ iff for all $t \in \mathsf{Tid}$, $m \in L.M$ and $\widetilde{v} \in \mathsf{Val}^*$, we have:

(1) $L \notin \Lambda$, and $I(t,m,\widetilde{v})$ only calls methods of the libraries of Λ.
(2) $\langle\langle -, k+1\rangle, -\rangle \notin [\![I(t,m,\widetilde{v})]\!]_t$, i.e. the implementation of a method call $m(\widetilde{v})$ cannot return with a non-zero break number, and thus cannot cause a loop containing a call to $m(\widetilde{v})$ to break inappropriately.
(3) if $\langle\langle v, 0\rangle, \langle E, \mathrm{po}\rangle\rangle \in [\![I(t,m,\widetilde{v})]\!]_t$ then $E \neq \emptyset$, i.e. if an implementation successfully executes, it must contain at least one method call.

We note $\mathtt{loc}(I)$ the set of all locations that can be accessed by the implementation of I: $\mathtt{loc}(I) \triangleq \bigcup_{t,m,\widetilde{v}} \bigcup_{(-,\langle E,-\rangle)\in[\![I(t,m,\widetilde{v})]\!]_t} \mathtt{loc}(E)$. We then define a function $[\![\_]\!]_I$ to map an implementation I to a concurrent program as follows.

$$[\![v]\!]_{t,I} \triangleq v \qquad [\![m(v_1,\ldots,v_k)]\!]_{t,I} \triangleq \begin{cases} I(t,m,\langle v_1,\ldots,v_k\rangle) & \text{if } m \in L.M \\ m(v_1,\ldots,v_k) & \text{otherwise} \end{cases}$$

$$[\![\mathtt{loop\ p}]\!]_{t,I} \triangleq \mathtt{loop}\ [\![\mathtt{p}]\!]_{t,I} \qquad [\![\mathtt{let\ p\ f}]\!]_{t,I} \triangleq \mathtt{let}\ [\![\mathtt{p}]\!]_{t,I}\ (\lambda v.[\![\mathtt{f}\ v]\!]_{t,I})$$

$$[\![\mathtt{break}_k\ v]\!]_{t,I} \triangleq \mathtt{break}_k\ v \qquad [\![\langle \mathtt{p}_1,\ldots,\mathtt{p}_T\rangle]\!]_I \triangleq \langle [\![\mathtt{p}_1]\!]_{1,L},\ldots,[\![\mathtt{p}_T]\!]_{T,L}\rangle$$

As an example, we can define the implementation I_{SV} of the broadcast library into $\text{RDMA}^{\text{WAIT}}$. For each location x of the broadcast library, we create a location x_n for each node $n \in \mathsf{Node}$. We also create a dummy location per node, $\bot_n$ for $n \in \mathsf{Node}$, and we use an additional dummy work identifier d_0.

$$\begin{aligned}
I_{\mathsf{SV}}(t,\mathtt{Write}_{\mathsf{sv}},(x,v)) &\triangleq \mathtt{Write}(x_{\mathsf{n}(t)},v)\\
I_{\mathsf{SV}}(t,\mathtt{Read}_{\mathsf{sv}},(x)) &\triangleq \mathtt{Read}(x_{\mathsf{n}(t)})\\
I_{\mathsf{SV}}(t,\mathtt{Bcast}_{\mathsf{sv}},(x,d,\{n_1,\ldots,n_k\})) &\triangleq \mathtt{Put}(x_{n_1},x_{\mathsf{n}(t)},d);\ldots;\mathtt{Put}(x_{n_k},x_{\mathsf{n}(t)},d)\\
I_{\mathsf{SV}}(t,\mathtt{Wait}_{\mathsf{sv}},(d)) &\triangleq \mathtt{Wait}(d)\\
I_{\mathsf{SV}}(t,\mathtt{GF}_{\mathsf{sv}},(\{n_1,\ldots,n_k\})) &\triangleq \mathtt{Get}(\bot_{\mathsf{n}(t)},\bot_{n_1},d_0);\ldots;\mathtt{Get}(\bot_{\mathsf{n}(t)},\bot_{n_k},d_0);\mathtt{Wait}(d_0)
\end{aligned}$$

where $\{\mathtt{Write},\mathtt{Read},\mathtt{Put},\mathtt{Get},\mathtt{Wait}\}$ are methods of the $\text{RDMA}^{\text{WAIT}}$ library (see §3.3).

A read/write on a thread t accesses the location of its node $\mathsf{n}(t)$. A broadcast executes multiple Put operations. Each of them reads the location of its node and overwrites the location of a designated

node. A wait operation works similarly to RDMA$^{\text{WAIT}}$. Finally, a global fence executes a `Get` operation towards each node requiring fencing, and waits for the completion of all the `Get` operations. As mentioned in the overview, this ensures that all previous NIC operations towards these nodes are completely finished.

We can easily see that I_{SV} is well defined, as it cannot return a break number greater than zero, and every (succeeding) implementation generates at least one event.

Using these definitions, we arrive at a notion of a sound implementation, which holds whenever the implementation is a refinement of the library specification.

Definition 3.10. We say that I is a *sound implementation* of L using Λ if, for any program $\widetilde{\mathsf{p}}$ such that $\mathtt{loc}(I) \cap \mathtt{loc}(\widetilde{\mathsf{p}}) = \emptyset$, we have that $\mathtt{outcome}_{\Lambda}(\llbracket \widetilde{\mathsf{p}} \rrbracket_I) \subseteq \mathtt{outcome}_{\Lambda \uplus \{L\}}(\widetilde{\mathsf{p}})$.

For a concurrent program $\widetilde{\mathsf{p}}$ using methods of $(\Lambda \uplus \{L\})$, $\llbracket \widetilde{\mathsf{p}} \rrbracket_I$ only uses methods of Λ. The implementation I is sound if the translation does not introduce any new outcomes. We can assume I and $\widetilde{\mathsf{p}}$ use disjoint locations to avoid capture of location names.

3.6 Abstractions and Locality

We now work towards the modular proof technique for verifying soundness of an implementation against a library in Mowgli. As is common in proofs of refinement, we use an *abstraction function* [Abadi and Lamport 1991] mapping the concrete implementation to its abstract library specification. For $f : A \to B$ and $r \subseteq A \times A$, we note $f(r) \triangleq \{\langle f(x), f(y)\rangle \mid \langle x, y\rangle \in r\}$.

Definition 3.11. Suppose I is a well-defined implementation of a library L using Λ, and that $G = \langle E, \mathrm{po}\rangle$ and $G' = \langle E', \mathrm{po}'\rangle$ are plain executions using methods of Λ and L respectively. We say that a surjective function $f : E \to E'$ abstracts G to G', denoted $\mathsf{abs}^f_{I,L}(G, G')$, iff

- $E|_L = \emptyset$ (i.e. G contains no calls to the abstract library L) and $E'|_L = E'$ (i.e. G' only contains calls to the abstract library L);
- $f(\mathrm{po}) \subseteq (\mathrm{po}')^*$ and $\forall \mathsf{e}_1, \mathsf{e}_2,\ \langle f(\mathsf{e}_1), f(\mathsf{e}_2)\rangle \in \mathrm{po}' \implies \langle \mathsf{e}_1, \mathsf{e}_2\rangle \in \mathrm{po}$; and
- if $\mathsf{e}' = \langle t, \iota, \langle m, \widetilde{v}, v'\rangle\rangle \in E'$ then $\langle \langle v', 0\rangle, G|_{f^{-1}(\mathsf{e}')}\rangle \in \llbracket I(t, m, \widetilde{v})\rrbracket_t$

Intuitively, $\mathsf{abs}^f_{I,L}(G, G')$ means there is some abstract concurrent program $\widetilde{\mathsf{p}}$ on library L such that $\langle \_, G'\rangle \in \llbracket \mathsf{p}\rrbracket$ is a plain execution of the abstract program, $\langle \_, G\rangle \in \llbracket \lfloor \mathsf{p} \rfloor_I \rrbracket$ is a plain execution of its implementation, and G and G' behave similarly. The abstraction function f maps every event of the implementation to the abstract method call it was created for. The second requirement states that the program order is preserved in both directions. The last requirement states that, for each abstract event e', its implementation $G|_{f^{-1}(\mathsf{e}')}$ behaves properly. We ask that this subgraph be a valid plain execution of the implementation with the same output value.

Lemma 3.12. *Given $\widetilde{\mathsf{p}}$ on library L and a well-defined implementation I of L, if $\langle \widetilde{v}, G\rangle \in \llbracket \lfloor \widetilde{\mathsf{p}} \rfloor_I \rrbracket$ then there exists $\langle \widetilde{v}, G'\rangle \in \llbracket \widetilde{\mathsf{p}}\rrbracket$ and f such that $\mathsf{abs}^f_{I,L}(G, G')$.*

Finally, we can define a notion of local soundness for an implementation.

Definition 3.13. We say that a well defined implementation I of a library L is *locally sound* iff, whenever we have a Λ-consistent execution $\mathcal{G} = \langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle$ and $\mathsf{abs}^f_{I,L}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$, then there exists $\mathtt{stmp}'$, so', and a concretisation function $g : \langle E', \mathrm{po}', \mathtt{stmp}'\rangle.\mathsf{SEvent} \to \mathcal{G}.\mathsf{SEvent}$ such that:

- $g(\langle \mathsf{e}', a'\rangle) = \langle \mathsf{e}, a\rangle$ implies $f(\mathsf{e}) = \mathsf{e}'$ and
  - For all a_0 such that $\langle a_0, a'\rangle \in \mathrm{to}$, there exists $\langle \mathsf{e}_1, a_1\rangle \in \mathcal{G}.\mathsf{SEvent}$ such that $f(\mathsf{e}_1) = \mathsf{e}'$, $\langle a_0, a_1\rangle \in \mathrm{to}$, and $\langle \langle \mathsf{e}_1, a_1\rangle, \langle \mathsf{e}, a\rangle\rangle \in \mathrm{hb}^*$;

  - For all a_0 such that $\langle a', a_0\rangle \in \mathsf{to}$, there exists $\langle \mathrm{e}_2, a_2\rangle \in \mathcal{G}.\mathrm{SEvent}$ such that $f(\mathrm{e}_2) = \mathrm{e}'$, $\langle a_2, a_0\rangle \in \mathsf{to}$, and $\langle\langle \mathrm{e}, a\rangle, \langle \mathrm{e}_2, a_2\rangle\rangle \in \mathsf{hb}^*$.
- $g(\mathsf{so}') \subseteq \mathsf{hb}$;
- For all hb' transitive such that $(\mathsf{ppo}' \cup \mathsf{so}')^+ \subseteq \mathsf{hb}'$ and $g(\mathsf{hb}') \subseteq \mathsf{hb}$, we have $\langle E', \mathrm{po}', \mathtt{stmp}', \mathsf{so}', \mathsf{hb}'\rangle \in L.\mathcal{C}$, where $\mathsf{ppo}' \triangleq \langle E', \mathrm{po}', \mathtt{stmp}'\rangle.\mathsf{ppo}$.

Unlike the notion of soundness (*cf.* Def. 3.10) expressed using an arbitrary program, local soundness is expressed using an arbitrary abstraction. It states that whenever we have an abstraction from $\langle E, \mathrm{po}\rangle$ to $\langle E', \mathrm{po}'\rangle$ and we know the implementation $\langle E, \mathrm{po}\rangle$ has a Λ-consistent execution $\mathcal{G}$, then the abstract plain execution $\langle E', \mathrm{po}'\rangle$ also has an L-consistent execution (third point) and the implementation respects the synchronisation promises made by the abstract library L (first and second point).

To translate the synchronisation promises, we require a *concretisation function* g that maps every subevent of the abstraction to a subevent in their implementation. The library L makes two kinds of synchronisation promises: to (via stamps) and so'. If we have $\langle \mathrm{s}'_1, \mathrm{s}'_2\rangle \in \mathsf{so}'$ in the abstraction, then we require that the concretisation of s'_1 synchronises with the concretisation of s'_2, i.e. we ask that $g(\mathsf{so}') \subseteq \mathsf{hb}$.

Whenever the abstraction contains a subevent of the form $\langle \mathrm{e}', a'\rangle$, the usage of the stamp a' carries an obligation. The subevent promises to synchronise with *any* earlier or later subevent, not necessarily from library L, according to the to relation (*cf.* Fig. 10 for RDMA). In most cases, the concretisation uses the same stamp, i.e. $g(\langle \mathrm{e}', a'\rangle) = \langle \mathrm{e}, a\rangle$ with $a' = a$, and the property is trivially respected by the implementation with $\langle \mathrm{e}_1, a_1\rangle = \langle \mathrm{e}_2, a_2\rangle = \langle \mathrm{e}, a\rangle$. Otherwise we have $a' \neq a$, and so for any earlier (resp. later) stamp a_0 that a' should synchronise with, we need to justify this synchronisation happens in the implementation, i.e. that we have $\langle \mathrm{e}_1, a_1\rangle \xrightarrow{\mathsf{hb}^*} \langle \mathrm{e}, a\rangle$, where a_1 can perform the expected stamp synchronisation $\langle a_0, a_1\rangle \in \mathsf{to}$.

An important point to note is that hb is potentially bigger than $(\mathsf{ppo} \cup \mathsf{so})^+$. In which case, we need to prove the result for any reasonable hb' bigger than $(\mathsf{ppo}' \cup \mathsf{so}')^+$. Thus local soundness states that if the implementation has a Λ-consistent execution *with additional constraints*, then the abstraction similarly has an L-consistent execution *with these additional constraints*. This is required for the implementation to work in any context, i.e. for programs using L in conjunction to other libraries, as expressed by the following theorem.

Theorem 3.14. *If a well-defined implementation is locally sound, then it is sound.*

Proof. See Theorem E.3. □

In the case of the shared variable library, we can use this proof technique to verify the implementation I_{SV}.

Theorem 3.15. *I_{SV} is locally sound, and hence I_{SV} is sound.*

Proof. See Theorem G.1. □

4 Barrier Library

As discussed informally in §2.2, LOCO implements a barrier library (BAL), which supports synchronisation of threads across multiple threads. Note that each barrier corresponds to a set of threads, which we refer to as the "participating threads" of a barrier. Each participating thread must wait for *all* operations towards *all* participating threads (including its own) that are po-before each barrier to be completed. We first present a generic specification for barriers with participating nodes in §4.1, and the LOCO barrier and its correctness proof in §4.2. In §4.3 we discuss an issue with such a barrier that only synchronises participating nodes and a possible fix.

4.1 Generic Barrier Specification

The barrier library (BAL) only has the single method $\mathsf{BAR}_{\text{BAL}} : \mathsf{Loc} \to ()$, taking a location as an input and producing no output. Thus, we have $\mathtt{loc}(\mathsf{BAR}_{\text{BAL}}(x)) = \{x\}$. The input location x defines the set of threads that synchronise via $\mathsf{BAR}_{\text{BAL}}(x)$. In our model, we assume a function $\mathsf{b} : \mathsf{Loc} \to \mathcal{P}(\mathsf{Tid})$ associating each location x with a set of threads that perform a barrier synchronisation on x.

While the LOCO barrier implementation (see §4.2) supports synchronisation across nodes connected by RDMA, our specification is more general and abstracts away the notion of nodes. Instead, our library defines synchronisation between *threads*, providing freedom to implement different synchronisation mechanisms depending on whether the threads are on the same or on different nodes.

Since Mowgli allows libraries to be defined in isolation, we only consider E containing barrier calls. Let $E_x \triangleq \{\mathsf{e} \in E \mid \mathtt{loc}(\mathsf{e}) = \{x\}\}$ denote the set of barrier calls on the location x.

Definition 4.1 (BAL-consistency). We say that $\mathcal{G} = \langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle$ is BAL-consistent iff:

- $\mathtt{stmp} = \mathtt{stmp}_{\mathsf{BAL}}$, defined as $\mathtt{stmp}_{\mathsf{BAL}}(\langle \_, \_, \langle \mathsf{BAR}_{\text{BAL}}, (x), ()\rangle\rangle) = \bigcup_{t \in \mathsf{b}(x)} \left\{\mathsf{aGF}_{\mathsf{n}(t)}\right\} \cup \{\mathsf{aCR}\}$;
- for all x and $\mathsf{e} \in E_x$, $\mathsf{t}(\mathsf{e}) \in \mathsf{b}(x)$; i.e. non-participating threads do not participate;
- for all $x \in \mathsf{Loc}$, there is an integer c_x such that for all thread $t \in \mathsf{b}(x)$ we have $\#(E_x|_t) = c_x$; i.e. each participating thread makes exactly c_x calls to the barrier on x;
- there is an ordering function $o : E \to \mathbb{N}$ such that for all location x:
  - if $\mathsf{e} \in E_x$ then $1 \le o(\mathsf{e}) \le c_x$;
  - if $\mathsf{e}_1, \mathsf{e}_2 \in E_x$ and $\langle \mathsf{e}_1, \mathsf{e}_2\rangle \in \mathrm{po}$ then $o(\mathsf{e}_1) < o(\mathsf{e}_2)$; and
- $\mathrm{so} = \bigcup_{x \in \mathsf{Loc}} \bigcup_{1 \le i \le c_x} \left\{\langle \langle \mathsf{e}_1, \mathsf{aGF}_n\rangle, \langle \mathsf{e}_2, \mathsf{aCR}\rangle\rangle \;\middle|\; \mathsf{e}_1, \mathsf{e}_2 \in (E_x \cap o^{-1}(i))\right\}$

This predicate clearly respects monotonicity (since hb is unrestricted) and decomposability (since each location is treated independently).

The function o associates each barrier call to the number of times the location has been used by this thread, in program order. We say that e_1 and e_2 *synchronise together* iff $\mathtt{loc}(\mathsf{e}_1) = \mathtt{loc}(\mathsf{e}_2)$ and $o(\mathsf{e}_1) = o(\mathsf{e}_2)$. The stamps of the form aGF correspond to the *entry points* of the barrier calls, waiting for previous operations to finish before the synchronisation. The stamp aCR represents the *exit point* of the barrier, after the synchronisation. The synchronisation is then an so ordering between aGF and aCR for barrier calls that synchronise together.

4.2 LOCO Implementation

Given $\mathsf{b} : \mathsf{Loc} \to \mathcal{P}(\mathsf{Tid})$, for each location x with $\mathsf{b}(x) = \{t_1, \ldots, t_k\}$ synchronising k threads, we create a set of k shared variables (i.e. sv locations) $\{x_{t_1}, \ldots, x_{t_k}\}$. Each shared variable x_t is used as a counter indicating how many times thread t has executed a barrier on x. The LOCO implementation decomposes the barrier into three steps: (1) wait for previous operations to finish; (2) increase your counter; (3) wait for the counters of other threads to increase. We define the implementation $I^{\mathsf{b}}_{\mathsf{BAL}}$ in Fig. 11. Clearly, the implementation is well defined: it cannot return a break number greater than zero, since all break commands have a break number of 1 and are inside loops; and every succeeding implementation generates at least one event.

For $t \notin \mathsf{b}(x)$: $I^{\mathsf{b}}_{\mathsf{BAL}}(t, \mathsf{BAR}_{\text{BAL}}, (x)) \triangleq \mathtt{loop}\ \{()\}$

For $t \in \mathsf{b}(x) = \{t_1, \ldots, t_k\}$:

$I^{\mathsf{b}}_{\mathsf{BAL}}(t, \mathsf{BAR}_{\text{BAL}}, (x)) \triangleq$

```
    let s_n = {n(t_i) | t_i ∈ b(x)} in
    GF_sv(s_n);
    let v = Read_sv(x_t) in
    Write_sv(x_t, v + 1);
    Bcast_sv(x_t, _, (s_n \ {n(t)}));
    loop {
        let v' = Read_sv(x_{t_1}) in
        if v' > v then break_1() else () };
    ...
    loop {
        let v' = Read_sv(x_{t_k}) in
        if v' > v then break_1() else () }
```

Fig. 11. $I^{\mathsf{b}}_{\mathsf{BAL}}$ implementation

If a method call is made by a non-participating thread, the call is invalid and we implement it using a non-terminating loop. This is necessary for soundness, as the outcomes of the implementation must be valid, and in this situation the BAL specification does not allow any valid outcomes.

If a method call is made by a participating thread t, the implementation starts with a global fence ensuring any previous operation towards any relevant node is fully finished. Then, it increments its counter x_t to indicate to other threads that the barrier has been reached and executed. The value of x_t is immediately available to other threads on the same node, and is made available to other participating nodes using a broadcast. Note that the broadcast does not perform a loopback (i.e. we exclude $\mathsf{n}(t)$ from the targets), as asking the NIC to overwrite x_t with itself might cause the new value of a later barrier call to be reverted to the current value. Then, we repeatedly read the (local) values of the other counters x_{t_i} and wait for each of them to indicate other threads have reached their matching barrier call. Note that there is no reason to wait for the broadcast to finish: the implementation on t might go ahead before other threads are aware that t reached the barrier, but that does not break the guarantees provided by the barrier.

THEOREM 4.2. *The implementation $I^{\mathsf{b}}_{\mathsf{BAL}}$ is locally sound.*

PROOF. See Theorem G.5. □

4.3 Supporting Transitivity

		$x = 0$
$\overline{x} := 1$	$\mathsf{BAR}_{\mathrm{BAL}}(b_1)$	$\mathsf{BAR}_{\mathrm{BAL}}(b_2)$
$\mathsf{BAR}_{\mathrm{BAL}}(b_1)$	$\mathsf{BAR}_{\mathrm{BAL}}(b_2)$	$a := x$

$a = 0$ ✓

Fig. 12. Allowed weak barrier behaviour

The barrier semantics in §4.1 only performs a global fence on nodes with participating threads. While this appears intuitive and reduces assumptions about other nodes, barrier synchronisation using such a library is *not* transitive. For example, consider the program in Fig. 12. Since $\overline{x} := 1$ is an operation towards node 3, the barrier $\mathsf{BAR}_{\mathrm{BAL}}(b_1)$ does not wait for it to finish, allowing $a = 0$.

Such a transitive barrier can straightforwardly be obtained by synchronising across *all* nodes, instead of just "participating" threads. For the specification, we define $\mathtt{stmp}_{\mathsf{BAL}}(\langle \_, \_, \langle \mathsf{BAR}_{\mathrm{BAL}}, (x), () \rangle \rangle) = \bigcup_{n \in \mathsf{Node}} \{\mathsf{aGF}_n\} \cup \{\mathsf{aCR}\}$ and for the implementation, we define $I^{\mathsf{b}}_{\mathsf{BAL}}(t, \mathsf{BAR}_{\mathrm{BAL}}, (x)) \triangleq \mathtt{let}\, s_n = \mathsf{Node}\ \mathtt{in}\ \ldots$. This stronger version is the one implemented in LOCO (see Fig. 7).

5 Ring Buffer Library

The ring buffer library (RBL) provides methods for a single-writer-multiple-reader FiFo queue for messages of any size, where each message is duplicated as necessary and can be read once by each reader. Here, we present its specification (§5.1), and an implementation and correctness proof (§5.2).

5.1 Ring Buffer Specification

The ring buffer library has two methods $\mathtt{Submit}^{\mathrm{RBL}} : \mathsf{Loc} \times \mathsf{Val}^* \to \mathbb{B}$ and $\mathtt{Receive}^{\mathrm{RBL}} : \mathsf{Loc} \to \mathsf{Val}^* \uplus \{\bot\}$, with $\mathtt{loc}(\mathtt{Submit}^{\mathrm{RBL}}(x, \_)) = \mathtt{loc}(\mathtt{Receive}^{\mathrm{RBL}}(x)) = \{x\}$. $\mathtt{Submit}^{\mathrm{RBL}}(x, \widetilde{v})$ tries to add a new message $\widetilde{v}$ to the ring buffer x. It can either fail if the ring buffer is full, returning `false`, or succeed returning `true`. $\mathtt{Receive}^{\mathrm{RBL}}(x)$ tries to read a message from the ring buffer x. It can either succeed if there is at least one pending message, returning the next message, or fail if there is no pending messages, returning $\bot$.

In our model, we assume two functions $\mathtt{wthd} : \mathsf{Loc} \to \mathsf{Tid}$ and $\mathtt{rthd} : \mathsf{Loc} \to \mathcal{P}(\mathsf{Tid})$ associating each location x with a writing thread $\mathtt{wthd}(x)$ and a set of reader threads $\mathtt{rthd}(x)$. For subevents, we define the stamping function $\mathtt{stmp}_{\mathsf{RBL}}$ as follows:

$$\mathtt{stmp}_{\mathsf{RBL}}(\langle t, \_, \langle \mathtt{Submit}^{\mathrm{RBL}}, (x, \_), \mathtt{true} \rangle \rangle) \triangleq \left\{ \mathsf{aNRW}_{\mathsf{n}(t')} \;\middle|\; t' \in \mathtt{rthd}(x) \wedge \mathsf{n}(t') \neq \mathsf{n}(t) \right\} \cup \{\mathsf{aCW}\}$$

$$\mathtt{stmp}_{\mathsf{RBL}}(\langle t, \_, \langle \mathtt{Submit}^{\mathrm{RBL}}, (x, \_), \mathtt{false} \rangle \rangle) \triangleq \{\mathsf{aWT}\}$$

$$\mathtt{stmp}_{\mathsf{RBL}}(\langle \_, \_, \langle \mathtt{Receive}^{\mathrm{RBL}}, (x), \widetilde{v}\rangle\rangle) \triangleq \{\mathtt{aCR}\}$$
$$\mathtt{stmp}_{\mathsf{RBL}}(\langle \_, \_, \langle \mathtt{Receive}^{\mathrm{RBL}}, (x), \bot\rangle\rangle) \triangleq \{\mathtt{aWT}\}$$

A successful call to $\mathtt{Submit}^{\mathrm{RBL}}$ (with return value `true`) is denoted by a write stamp for each relevant node: the stamp aCW is used by the writer node, and the stamps $\mathtt{aNRW}_{\mathsf{n}(t')}$ are used by the corresponding remote nodes. Failing calls (with return value `false` or $\bot$) are depicted by the stamp aWT. Finally, a succeeding $\mathtt{Receive}^{\mathrm{RBL}}$ call uses the reading stamp aCR.

We note different sets corresponding to calls to $\mathtt{Submit}^{\mathrm{RBL}}$ succeeding ($\mathcal{W}$) and calls to $\mathtt{Receive}^{\mathrm{RBL}}$ failing ($\mathcal{F}$) or succeeding ($\mathcal{R}$). Calls to $\mathtt{Submit}^{\mathrm{RBL}}$ failing are ignored by the specification.

$$\begin{aligned}
\mathcal{W}_x^n &\triangleq \left\{\langle \mathsf{e}, \mathtt{aNRW}_n\rangle \;\middle|\; \mathsf{e} = \langle t, _, \langle \mathtt{Submit}^{\mathrm{RBL}}, (x, _), \mathtt{true}\rangle\rangle \in E \wedge \mathtt{aNRW}_n \in \mathtt{stmp}_{\mathsf{RBL}}(\mathsf{e})\right\} \\
&\quad \cup \left\{\langle \mathsf{e}, \mathtt{aCW}\rangle \;\middle|\; \mathsf{e} = \langle t, _, \langle \mathtt{Submit}^{\mathrm{RBL}}, (x, _), \mathtt{true}\rangle\rangle \in E \wedge \mathsf{n}(t) = n\right\} \\
\mathcal{F}_x^n &\triangleq \left\{\langle \mathsf{e}, \mathtt{aWT}\rangle \;\middle|\; \mathsf{e} = \langle t, _, \langle \mathtt{Receive}^{\mathrm{RBL}}, (x), \bot\rangle\rangle \in E \wedge \mathsf{n}(t) = n\right\} \\
\mathcal{R}_x^n &\triangleq \left\{\langle \mathsf{e}, \mathtt{aCR}\rangle \;\middle|\; \mathsf{e} = \langle t, _, \langle \mathtt{Receive}^{\mathrm{RBL}}, (x), \widetilde{v}\rangle\rangle \in E \wedge \mathsf{n}(t) = n\right\}
\end{aligned}$$

We then define the reads-from relation rf matching successful $\mathtt{Submit}^{\mathrm{RBL}}$ and $\mathtt{Receive}^{\mathrm{RBL}}$ events.

Definition 5.1. Given $\mathcal{G} = \langle E, \mathrm{po}, \mathtt{stmp}_{\mathsf{RBL}}, \_, \_\rangle$, we say that rf is *well-formed* iff each of the following holds:

(1) $\mathrm{rf} = \bigcup_{n,x} \mathrm{rf}_x^n$ with $\mathrm{rf}_x^n \subseteq \mathcal{W}_x^n \times \mathcal{R}_x^n$
(2) rf_x^n is total and functional on its range, i.e. each read subevent in $\mathcal{R}_x^n$ is related to exactly one write subevent in $\mathcal{W}_x^n$.
(3) If $(\langle \_, \_, \langle \mathtt{Submit}^{\mathrm{RBL}}, (x, \widetilde{v}), \mathtt{true}\rangle\rangle, a) \xrightarrow{\mathrm{rf}} (\langle \_, \_, \langle \mathtt{Receive}^{\mathrm{RBL}}, (x), \widetilde{v}'\rangle\rangle, a')$ then $\widetilde{v} = \widetilde{v}'$, i.e. related events write and read the same tuple of values.
(4) If $\langle \mathsf{s}_1, \mathsf{s}_2\rangle \in \mathrm{rf}$, $\langle \mathsf{s}_1, \mathsf{s}_3\rangle \in \mathrm{rf}$, and $s_2 \neq s_3$, then $\mathtt{t}(\mathsf{s}_2) \neq \mathtt{t}(\mathsf{s}_3)$, i.e. each thread can read each message at most once.
(5) If $\mathsf{s}_1, \mathsf{s}_2 \in \mathcal{W}_x^n$, $\langle \mathsf{s}_1, \mathsf{s}_2\rangle \in \mathrm{po}$, and $\langle \mathsf{s}_2, \mathsf{s}_4\rangle \in \mathrm{rf}$, then there is s_3 such that $\langle \mathsf{s}_1, \mathsf{s}_3\rangle \in \mathrm{rf}$, and $\langle \mathsf{s}_3, \mathsf{s}_4\rangle \in \mathrm{po}$, i.e. threads cannot jump a message.

We define the *fails-before* relation fb expressing that a failing $\mathtt{Receive}^{\mathrm{RBL}}$ occurs before a succeeding $\mathtt{Submit}^{\mathrm{RBL}}$ as follows:

$$\mathrm{fb} \triangleq \bigcup_{n,x} \left(\mathcal{F}_x^n \times \mathcal{W}_x^n \setminus (\mathrm{po}^{-1}; \mathrm{rf}^{-1})\right)$$

If $\mathsf{s}_1 \in \mathcal{W}_x^n$ and $\mathsf{s}_3 \in \mathcal{F}_x^n$, then the contents written by s_1 is not available when s_3 is executed. Either there is s_2 such that $\langle \mathsf{s}_1, \mathsf{s}_2\rangle \in \mathrm{rf}$ and $\langle \mathsf{s}_2, \mathsf{s}_3\rangle \in \mathrm{po}$, in which case the message has been read; or there is no such s_2 and we have $\langle \mathsf{s}_3, \mathsf{s}_1\rangle \in \mathrm{fb}$ to express that the message was not yet written.

Definition 5.2 (RBL-consistency). We say that an execution $\mathcal{G} = \langle E, \mathrm{po}, \mathtt{stmp}_{\mathsf{RBL}}, \mathrm{so}, \mathrm{hb}\rangle$ is BAL-consistent iff:

- if $\langle t, \_, \langle \mathtt{Submit}^{\mathrm{RBL}}, (x, \_), \_\rangle\rangle \in E$ then $t = \mathtt{wthd}(x)$; and if $\langle t, \_, \langle \mathtt{Receive}^{\mathrm{RBL}}, (x), \_\rangle\rangle \in E$ then $t \in \mathtt{rthd}(x)$; and
- there exists a well-formed rf such that $\mathrm{so} = \mathrm{rf} \cup \mathrm{fb}$.

Note that this definition allows the writer thread to also be a reader, and nodes to have multiple reading threads. Moreover, the consistency predicate does not tell us anything about failing writes; they may fail spuriously.

$$
\begin{array}{l}
\texttt{For } \mathsf{wthd}(x) = t \wedge \mathsf{rthd}(x) = \{t_1; \ldots; t_k\} : \\
I^{\mathsf{wthd},\mathsf{rthd}}_{\mathsf{S},\mathsf{RBL}}(t, \mathsf{Submit}^{\mathrm{RBL}}, (x, \widetilde{v} = (v_1, \ldots, v_V)) \triangleq \\
\texttt{let } s_n = \{\mathsf{n}(t_i) \mid t_i \in \mathsf{rthd}(x)\} \setminus \{\mathsf{n}(t)\} \texttt{ in} \\
\texttt{let } V = \mathsf{len}(\widetilde{v}) \texttt{ in} \\
\texttt{let } H = \mathsf{Read}_{\mathrm{sv}}(h^x) \texttt{ in} \\
\texttt{let } H_1 = \mathsf{Read}_{\mathrm{sv}}(h^x_{t_1}) \texttt{ in} \\
\ldots \\
\texttt{let } H_k = \mathsf{Read}_{\mathrm{sv}}(h^x_{t_k}) \texttt{ in} \\
\texttt{let } M = \mathsf{min}(\{H_1, \ldots, H_k\}) \texttt{ in} \\
\texttt{if } (H - M) + (V + 1) > S \texttt{ then false else } \{ \\
\quad \mathsf{Write}_{\mathrm{sv}}(x_{H\%S}, V);\ \mathsf{Bcast}_{\mathrm{sv}}(x_{H\%S}, \_, s_n); \\
\quad \mathsf{Write}_{\mathrm{sv}}(x_{(H+1)\%S}, v_1);\ \mathsf{Bcast}_{\mathrm{sv}}(x_{(H+1)\%S}, \_, s_n); \\
\quad \ldots \\
\quad \mathsf{Write}_{\mathrm{sv}}(x_{(H+V)\%S}, v_V);\ \mathsf{Bcast}_{\mathrm{sv}}(x_{(H+V)\%S}, \_, s_n); \\
\quad \mathsf{Wait}_{\mathrm{sv}}(d_x); \\
\quad \mathsf{Write}_{\mathrm{sv}}(h^x, H + V + 1);\ \mathsf{Bcast}_{\mathrm{sv}}(h^x, d_x, s_n); \\
\quad \texttt{true } \};
\end{array}
$$

$$
\begin{array}{l}
\texttt{For } t \notin \mathsf{rthd}(x): \\
I^{\mathsf{wthd},\mathsf{rthd}}_{\mathsf{S},\mathsf{RBL}}(t, \mathsf{Receive}^{\mathrm{RBL}}, (x)) \triangleq \texttt{loop } \{()\} \\
\\
\texttt{For } t \in \mathsf{rthd}(x): \\
I^{\mathsf{wthd},\mathsf{rthd}}_{\mathsf{S},\mathsf{RBL}}(t, \mathsf{Receive}^{\mathrm{RBL}}, (x)) \triangleq \\
\texttt{let } H = \mathsf{Read}_{\mathrm{sv}}(h^x_t) \texttt{ in} \\
\texttt{let } H' = \mathsf{Read}_{\mathrm{sv}}(h^x) \texttt{ in} \\
\texttt{if } H \geq H' \texttt{ then } \bot \texttt{ else } \{ \\
\quad \texttt{let } V = \mathsf{Read}_{\mathrm{sv}}(x_{H\%S}) \texttt{ in} \\
\quad \texttt{let } v_1 = \mathsf{Read}_{\mathrm{sv}}(x_{(H+1)\%S}) \texttt{ in} \\
\quad \ldots \\
\quad \texttt{let } v_V = \mathsf{Read}_{\mathrm{sv}}(x_{(H+V)\%S}) \texttt{ in} \\
\quad \mathsf{Write}_{\mathrm{sv}}(h^x_t, H + V + 1); \\
\quad \texttt{if } \mathsf{n}(\mathsf{wthd}(x)) = \mathsf{n}(t) \texttt{ then } () \texttt{ else} \\
\qquad \{ \mathsf{Bcast}_{\mathrm{sv}}(h^x_t, \_, \{\mathsf{n}(\mathsf{wthd}(x))\}) \}; \\
\quad (v_1, \ldots, v_V) \ \};
\end{array}
$$

Fig. 14. Implementation $I^{\mathsf{wthd},\mathsf{rthd}}_{\mathsf{S},\mathsf{RBL}}$ of the ring buffer library into sv

$a := \mathsf{Submit}^{\mathrm{RBL}}(x, 1)$	$b := \mathsf{Submit}^{\mathrm{RBL}}(y, 1)$
$\mathsf{GF}_{\mathrm{sv}}(\{n_2\})$	$\mathsf{GF}_{\mathrm{sv}}(\{n_1\})$
$c := \mathsf{Receive}^{\mathrm{RBL}}(y)$	$d := \mathsf{Receive}^{\mathrm{RBL}}(x)$

$(a, b, c, d) = (\mathsf{true}, \mathsf{true}, \bot, \bot)$ ✗

Fig. 13. Alternative ring buffer semantics

Alternative weaker semantics. Instead of requiring so = rf ∪ fb, we could give an alternative specification with so = rf and $\mathsf{hb}^{-1} \cap \mathsf{fb} = \emptyset$. The latter says that you still cannot ignore (fb) a write that you know (hb) has finished; but if you do ignore a write, you do not have to export the guarantee (so) that the write has not finished. For instance, take the litmus test in Fig. 13. With the semantics in Def. 5.2, at least one of the two $\mathsf{Receive}^{\mathrm{RBL}}$ has to succeed. With the weaker semantics, they are allowed to both fail, even when both $\mathsf{Submit}^{\mathrm{RBL}}$ calls succeed.

5.2 LOCO Implementation

As before, we assume given the functions $\mathsf{wthd} : \mathsf{Loc} \to \mathsf{Tid}$ and $\mathsf{rthd} : \mathsf{Loc} \to \mathcal{P}(\mathsf{Tid})$. We also assume an integer S representing the size of the ring buffer. We implement the ring buffer library (RBL) using the shared variable library (SV). For each location x with $\mathsf{rthd}(x) = \{t_1, \ldots, t_k\}$ we create the shared variable (i.e. SV locations) $x_0, \ldots, x_{S-1}$ for the content of the buffer, as well as shared variables h^x for the writer and $h^x_{t_1}; \ldots; h^x_{t_k}$ for the readers. We also use a work identifier d_x.

Events that do not respect rthd or wthd are implemented using an infinite loop (i.e. loop {()}), similarly to other implementations. Otherwise, we use the implementation $I^{\mathsf{wthd},\mathsf{rthd}}_{\mathsf{S},\mathsf{RBL}}$ given in Fig. 14, where % represents the modulo operation.

The value of h^x represents the next place to write for the writing thread. The value of $h^x_{t_i}$ represents the next place thread t_i needs to read. If $h^x = h^x_{t_i}$ then thread t_i is up-to-date and needs to wait for the writer to send additional data. If the difference between h^x and $h^x_{t_i}$ gets close to S, then the buffer is full and the writer cannot send any more data.

In the implementation of $\mathsf{Submit}^{\mathrm{RBL}}$, the value M represents the minimum of all $h^x_{t_i}$. As such, $(H - M)$ represents the amount of space currently in use. Since $(V + 1)$ represents the number of cells necessary to submit a new message (the size V itself is also submitted), we can proceed if $H - M + V + 1 \leq S$, i.e. if there is enough free space.

Since, for a specific remote node, the broadcasts complete in order, when a reader sees the new value of h^x it means the written data is available. We need to take care that the broadcast of h^x must read from the write of the *same* function call, and not from the write of a later call to $\texttt{Submit}^{\text{RBL}}$. Otherwise, the value of h^x for the second submit might be available to readers before the data of the second submit. For this, we simply need to wait for the broadcast of previous function calls, using $\texttt{Wait}_{\text{sv}}(d_x)$, before modifying h^x.

When thread t_i wants to receive, it only proceeds if $h^x > h^x_{t_i}$, otherwise t_i is up-to-date and returns $\perp$. After reading a message, the reader updates $h^x_{t_i}$ to signal to the writer the space of the message is no longer in use. If the reader is on the same node as the writer, there is no need for a broadcast, otherwise the reader broadcasts to the node of the writer.

With this implementation, each participating node possesses only one copy of the data, and potentially multiple readers per node can read from the same memory locations.

Theorem 5.3. *The implementation* $I^{\texttt{wthd,rthd}}_{\text{S,RBL}}$ *is locally sound.*

Proof. See Theorem G.7. □

6 Evaluation

In this section, we explore the performance of our LOCO primitives, then use them to build a high performance key-value store. Further applications can be found in §B.

All results were collected using `c6525 − 25g` nodes on the Cloudlab platform [clo [n. d.]]. These machines each have a 16-core AMD 7302P CPU, running Ubuntu 22.04. Nodes communicate over a 25 Gbps Ethernet fabric using Mellanox ConnectX-5 NICs.

6.1 LOCO Primitives

First, we compare the performance of the verified barrier (BAL) and ring buffer (RBL) primitives to equivalent operations in OpenMPI [Gabriel et al. 2004], a message-passing library commonly used to build distributed applications. We compare against OpenMPI 5.0.5, using the PML/UCX backend for RoCE support. Results are shown in Figure 15.

For the barrier experiments, we compare to the `MPI_Barrier` operation, varying both thread count per node and node count. The MPI barrier does not actually provide synchronization, expecting the user to instead appropriately track and fence operations before using the primitive. We compare the barrier to our LOCO barrier, both with and without the synchronization fence, and show that the LOCO barrier with equivalent semantics (no fence) performs as well or better than the MPI barrier. Note the MPI barrier dynamically switches between several internal algorithms adjusting to load leading to non-smooth performance across the test domain.

For the ring buffer experiments, we compare a ring buffer broadcast to the `MPI_Ibcast` (non-blocking broadcast) operation. We measure across different node counts and amounts of "network load", that is, the number n of outstanding broadcast operations in the network, along with total node count. A single node acts as the sender: it starts by sending n broadcasts, then sends a new one every time a prior message completes. All other nodes wait to receive and acknowledge messages. Messages have a fixed size of 64 bytes. Here, we find that the formally verified LOCO ring buffer provides better broadcast performance than MPI in most configurations, with MPI performance falling drastically as the number of outstanding messages rises.

6.2 Example Application: A Key-Value Store

Beyond our microbenchmarks, we describe an example LOCO application: a key-value store, built using composable LOCO primitives.

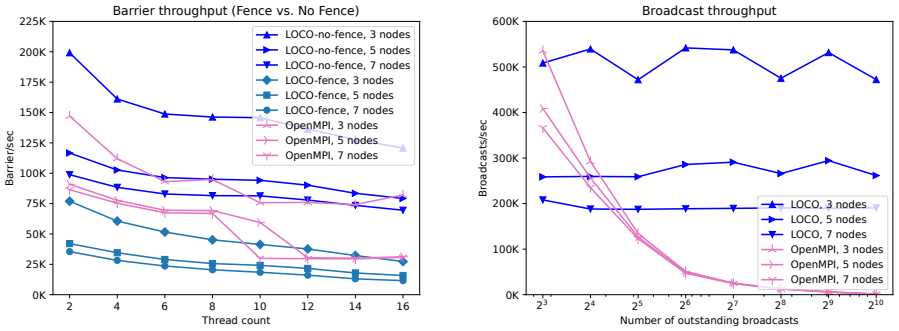


Fig. 15. Comparison of barrier and broadcast operations for LOCO and OpenMPI.

Our `kvstore` object is a distributed key-value store with a lookup operation that takes no locks, and insertion, deletion, and update operations protected by locks. Lookup and update are depicted in Fig. 16. Each node allocates a remotely-accessible memory region that is used to store values and consistency metadata (a checksum for atomicity, a counter for garbage collection, and a valid bit).

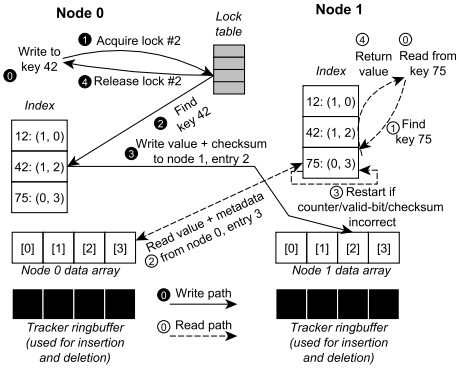


Fig. 16. `kvstore` read and write operations

Each node also maintains a local index (a C++ `unordered_map`), protected by a local reader-writer lock, which records the locations of all keys in the `kvstore` as (`node_id`, `array_index`) pairs, along with a counter matching the one stored with the data. The `kvstore` is linearisable, with a proof given in §H – our proof is simplified by leveraging the compositional properties of LOCO. Note that RDMA^{TSO} does not have a semantics for locks or RDMA read-modify-write operations, which means that this proof currently does not use Mowgli. We consider an extension of RDMA^{TSO} with synchronisation operations (and hence a full proof of `kvstore`) to be future work. Almost all RDMA maps [Barthels et al. 2015; Kalia et al. 2014; Li et al. 2023; Lu et al. 2024; Wang et al. 2022] lack any formal safety specification (we are only aware of two [Dragojević et al. 2014],[Alquraan et al. 2024]), likely due to difficulties in encapsulation, which the LOCO philosophy solves.

We compared our key-value store design against Sherman [she [n. d.]; Wang et al. 2022] and the MicroDB from Scythe [scy [n. d.]; Lu et al. 2024], two state-of-the-art RDMA key-value stores. We also compare against Redis-cluster [Ltd. 2021] as a non-RDMA baseline. Results are shown in Figure 17. We measured throughput on read-only, mixed read-write, and write-only operation distributions, across both uniform and Zipfian ($\theta = 0.99$) key distributions, and across different node counts and per-node thread counts. Each data point is the geometric mean of 5 runs with a 20 second duration, not including prefill.

All benchmarks use a 10MB keyspace, filled to 80% capacity with 64-bit keys and values. All benchmarks use the CityHash64 key hashing function [Pike and Alakuijala [n. d.]], and the YCSB-C implementation of a Zipfian distribution [ycs [n. d.]].

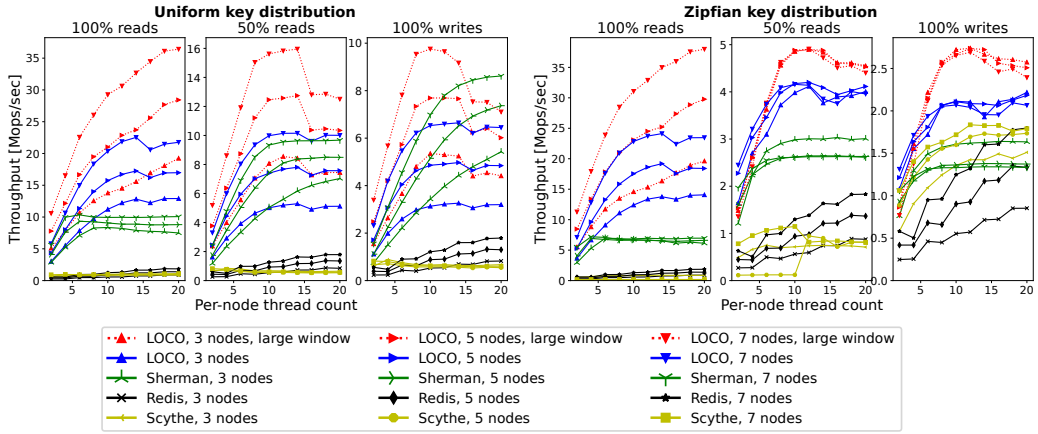


Fig. 17. Throughput comparison of key-value stores.

We modified Sherman to issue a fence (GF_{sv}) between lock-protected writes and lock releases to solve a bug related to consistency issues. Our `kvstore` also issues a fence for the same reason. For both, this fence incurs a 15% overhead.

For LOCO, Sherman, and Redis, write operations are updates. For Scythe, we found that stressing update operations led to program instability and very low throughput, so we use the performance of insertion operations as an upper bound on write performance. For Redis, we configure a cluster with no replication or persistence. Since each Redis server instance uses 4 threads, we create `ceil(num_threads/4)` server instances for a given thread count. We use Memtier [Ltd. 2024] as a benchmark client. Each node runs a single Memtier instance with threads equal to the thread count, and 128 clients per thread (matching the LOCO large window size).

In addition, all systems expose a parameter we call the *window size*, which specifies the maximum number of outstanding operations per application thread (note this is not a batch size – each operation is started and completed individually). Increasing LOCO's window size to 128 yielded significant improvement (the "large window" series). However, increasing Sherman's and Scythe's window sizes appeared to cause internal errors, so the main results for all systems except Redis (see above) use a window size of 3 for accurate comparison.

LOCO outperforms Sherman on read-only configurations. We believe this is because Sherman reads whole sections of the tree from remote memory, while the LOCO design looks up the location locally and only remotely reads the value. On the other hand, LOCO's advantage over Sherman for Zipfian writes likely comes from the better performance under contention.

Sherman outperforms LOCO (with a window size of 3) on mixed read-write and write-only distributions on uniform keys, while the reverse is true for Zipfian keys. Sherman's advantage here is likely due to the fact that, unlike LOCO, Sherman colocates locks with data, allowing them to issue lock releases in a batch with writes.

7 Related and Future Work

Although the formal semantics of RDMA has only recently been established [Ambal et al. 2024], our work is able to take advantage of earlier results in weak memory hardware [Alglave et al. 2014; Flur et al. 2016] and programming languages [Batty et al. 2011; Lahav et al. 2017]. We do not provide their details here since they are rather expansive.

RDMA Semantics. Prior works on RDMA semantics include coreRMA [Dan et al. 2016] (which formalises RDMA over the SC memory model) and RDMA^{TSO} [Ambal et al. 2024], a more realistic formal model that is very close to the Verbs library [linux-rdma 2018], describing the behaviour of RDMA over TSO. These semantics are however low-level and are difficult for programmers to use directly, as illustrated by examples such as those in Fig. 2.

RDMA Libraries. Much prior work in RDMA focuses on *upper-level primitives*, e.g. consensus protocols [Aguilera et al. 2019, 2020; Izraelevitz et al. 2023; Jha et al. 2019; Poke and Hoefler 2015], distributed maps or databases [Alquraan et al. 2024; Barthels et al. 2015; Dragojević et al. 2014, 2015; Gavrielatos et al. 2020; Kalia et al. 2014; Li et al. 2023; Wang et al. 2022], graph processing [Wang et al. 2023a], distributed learning [Ren et al. 2017; Xue et al. 2019], stand-alone data structures [Brock et al. 2019; Devarajan et al. 2020], disaggregated scheduling [Ruan et al. 2023a,b] or file systems [Yang et al. 2019, 2020]. These works focus on the final application, rather than considering the programming model as its own, partitionable problem. As a result, the intermediate library between RDMA and the exported primitive is usually ad-hoc and tightly coupled to the application, or effectively non-existent. In general, these applied, specific, projects manage raw memory explicitly statically allocated to particular nodes, use ad-hoc atomicity and consistency mechanisms, and do not consider the possibility of primitive reuse. This design is not a fundamentally flawed approach, but it does raise the possibility of a better mechanism, which likely could underlie all the above solutions.

Some works have considered this intermediate layer explicitly, however, the general approach for this intermediate layer has been to encapsulate local and remote memory as *distributed shared memory*, that is, a flat, uniform, coherent, and consistent address space hiding the relaxed consistency and non-uniform performance of the underlying RDMA network. These works generally focus on transparently (or mostly-transparently [Ruan et al. 2020; Zhang et al. 2022]) porting existing shared memory applications. We argue that this technique, either with purely software-based virtualisation [Cai et al. 2018; Gouk et al. 2022; Ruan et al. 2020; Wang et al. 2020; Zhang et al. 2022], or by extending hardware [Calciu et al. 2021], is unlikely to gain traction because the performance will always be worse than an approach which takes into account the underlying memory network.

Other programming models have simply used RDMA to implement existing distributed system abstractions. For example, both MPI [Message Passing Interface Forum 2023] and NCCL [NVIDIA Corporation 2020] can use RDMA for inter-node communication. However, fundamentally, these are *message passing programming models* with explicit send and receive primitives. While MPI does support some remote memory accesses, this support is best seen as a zero-copy send/receive mechanism where synchronisation is either coarse-grained and inflexible, or simply nonexistent. While message-passing is well-suited for dataflow applications (e.g. machine learning and signal processing) and highly parallel scale-out workloads (e.g. physical simulation), it is less useful for workloads that exhibit data-dependent communication [Liu et al. 2021], such as transaction processing or graph computations. In these applications, cross-node synchronisation is unavoidable and unpredictable, so the ideal performance strategy shifts from simply avoiding synchronisation to minimising contention, accelerating synchronisation use, and reducing data movement.

Compared to prior art, LOCO aims to build composable, reusable, and performant primitives for complicated memory networks, suitable for irregular workloads. No such option currently exists in the literature.

Verification. Our proofs have followed the declarative style [Raad et al. 2019; Stefanesco et al. 2024] enabling modular verification. RDMA^{TSO} [Ambal et al. 2024] also includes an operational model, which could form a basis for a program logic (e.g., [Bila et al. 2022; Lahav et al. 2023]), ultimately enabling operational abstractions and proofs of refinement [Dalvandi and Dongol 2022]. Other modular approaches include modular proofs through separation logics [Jung et al. 2018],

but this additionally requires a separation logic encoding of the RDMA$^{\text{WAIT}}$ memory model (and an associated proof of soundness) before it can be applied to verify libraries such as LOCO. We consider operational proofs and those involving separation logic as a topic for future work.

Nagasamudram et al. [2024] have verified, in Rocq, key properties of a coordination service known as Derecho [Jha et al. 2018], which can be configured to run over RDMA. However, their proofs start with a very high-level model called a *shared-state table*, which is an array of shared variables (*cf.* Fig. 7). Unlike our work, these assumed shared state table semantics have not been connected to any formal RDMA semantics. In future work, it would be interesting to connect our work to middleware such as Derecho, ultimately leading to a fully verified RDMA application stack.

There is a rich literature of work around model checking under weak and persistent memory [Abdulla et al. 2023; Kokologiannakis and Vafeiadis 2021] including recent works that tackle refinement and linearisability [Golovin et al. 2025; Raad et al. 2024]. It would be interesting to know whether these techniques can be extended to support RDMA$^{\text{TSO}}$ (and by extension RDMA$^{\text{WAIT}}$).

8 Conclusion

In this paper, we describe LOCO, a verified library for building composable and reusable objects in network memory and its associated proof system Mowgli. Our results show that LOCO can expose the full performance of underlying network memory to applications, while simultaneously easing proof burden.

Acknowledgments

This work was partially funded by industry partner Genuen, which provides hardware validation services using the harness described in Section B.2. Izraelevitz also privately contracted with this company to assist with commercialization efforts of this harness. Ambal is supported by the EPSRC grant EP/X037029/1 and Raad is supported by a UKRI fellowship MR/V024299/1, by the EPSRC grant EP/X037029/1, and by VeTSS. Dongol and Chockler are supported by EPSRC grants EP/Y036425/1, EP/X037142/1 and EP/X015149/1 and Royal Society grant IES\R1\221226. Dongol is additionally supported by EPSRC grant EP/V038915/1 and VeTSS. Vafeiadis is supported by ERC Consolidator Grant for the project "PERSIST" (grant agreement No. 101003349).

References

[n. d.]. The CloudLab Manual: Hardware. ([n. d.]). http://docs.cloudlab.us/hardware.html.

[n. d.]. RDMA core userspace libraries and daemons (rdma-core). ([n. d.]). https://github.com/linux-rdma/rdma-core.

[n. d.]. Scythe. ([n. d.]). https://github.com/PDS-Lab/Scythe.

[n. d.]. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. ([n. d.]). https://github.com/thustorage/Sherman.

[n. d.]. Yahoo! Cloud Serving Benchmark in C++. ([n. d.]). https://github.com/basicthinker/YCSB-C.

2014. *InfiniBand™ Architecture Specification Release 1.2.1 Annex A17: RoCEv2.* Technical Report Annex A17. InfiniBand™ Trade Association.

Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (1991), 253–284. https://doi.org/10.1016/0304-3975(91)90224-P

Parosh Aziz Abdulla, Mohamed Faouzi Atig, S. Krishna, Ashutosh Gupta, and Omkar Tuppe. 2023. Optimal Stateless Model Checking for Causal Consistency. In *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part I (Lecture Notes in Computer Science)*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.), Vol. 13993. Springer, 105–125. https://doi.org/10.1007/978-3-031-30823-9_6

Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. 2019. The Impact of RDMA on Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC '19)*. Association for Computing Machinery, New York, NY, USA, 409–418. https://doi.org/10.1145/3293611.3331601

Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. 2020. Microsecond Consensus for Microsecond Applications. In *14th USENIX Symposium on Operating Systems Design*

and Implementation (OSDI 20). USENIX Association, 599–616. https://www.usenix.org/conference/osdi20/presentation/aguilera

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. https://doi.org/10.1145/2627752

Ahmed Alquraan, Sreeharsha Udayashankar, Virendra Marathe, Bernardo Wong, and Samer Al-Kiswany. 2024. LoLKV: the logless, line the logless, linearizable, RDMA-based key-value storage system arizable, RDMA-based key-value storage system. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*. USENIX Association, USA, Article 3, 14 pages.

Guillaume Ambal, Brijesh Dongol, Haggai Eran, Vasileios Klimis, Ori Lahav, and Azalea Raad. 2024. Semantics of Remote Direct Memory Access: Operational and Declarative Models of RDMA on TSO Architectures. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 1982–2009. https://doi.org/10.1145/3689781

Andrew W. Appel and Sandrine Blazy. 2007. Separation Logic for Small-Step cminor. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings (Lecture Notes in Computer Science)*, Klaus Schneider and Jens Brandt (Eds.), Vol. 4732. Springer, 5–21. https://doi.org/10.1007/978-3-540-74591-4_3

Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing using RDMA. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1463–1475. https://doi.org/10.1145/2723372.2750547

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 55–66. https://doi.org/10.1145/1926385.1926394

Eleni Vafeiadi Bila, Brijesh Dongol, Ori Lahav, Azalea Raad, and John Wickerson. 2022. View-Based Owicki–Gries Reasoning for Persistent x86-TSO. In *Programming Languages and Systems*, Ilya Sergey (Ed.). Springer International Publishing, Cham, 234–261.

Benjamin Brock, Aydın Buluç, and Katherine Yelick. 2019. BCL: A Cross-Platform Distributed Data Structures Library. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP '19)*. Association for Computing Machinery, New York, NY, USA, Article 102, 10 pages. https://doi.org/10.1145/3337821.3337912

Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient distributed memory management with RDMA and caching. *Proc. VLDB Endow.* 11, 11 (jul 2018), 1604–1617. https://doi.org/10.14778/3236187.3236209

Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 79–92. https://doi.org/10.1145/3445814.3446713

J. Bradley Chen, Anita Borg, and Norman P. Jouppi. 1992. A simulation based study of TLB performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*. Association for Computing Machinery, New York, NY, USA, 114–123. https://doi.org/10.1145/139669.139708

Luca Corradini, Dragan Maksimovic, Paolo Mattavelli, and Regan Zane. 2015. *Digital control of high-frequency switched-mode power converters*. John Wiley & Sons.

Sadegh Dalvandi and Brijesh Dongol. 2022. Implementing and verifying release-acquire transactional memory in C11. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1817–1844. https://doi.org/10.1145/3563352

Andrei Marian Dan, Patrick Lam, Torsten Hoefler, and Martin Vechev. 2016. Modeling and Analysis of Remote Memory Access Programming. *SIGPLAN Not.* 51, 10 (oct 2016), 129–144. https://doi.org/10.1145/3022671.2984033

Hariharan Devarajan, Anthony Kougkas, Keith Bateman, and Xian-He Sun. 2020. HCL: Distributing Parallel Data Structures in Extreme Scales. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. 248–258. https://doi.org/10.1109/CLUSTER49012.2020.00035

Brijesh Dongol, Radha Jagadeesan, James Riely, and Alasdair Armstrong. 2018. On abstraction and compositionality for weak-memory linearisability. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings (Lecture Notes in Computer Science)*, Isil Dillig and Jens Palsberg (Eds.), Vol. 10747. Springer, 183–204. https://doi.org/10.1007/978-3-319-73721-8_9

Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 401–414. http://dl.acm.org/citation.cfm?id=2616448.2616486

Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 608–621. https://doi.org/10.1145/2837614.2837615

Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, 97–104.

Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. 2024. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 57–70.

Vasilis Gavrielatos, Antonios Katsarakis, Vijay Nagarajan, Boris Grot, and Arpit Joshi. 2020. Kite: efficient and available release consistency for the datacenter. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/3332466.3374516

Pavel Golovin, Michalis Kokologiannakis, and Viktor Vafeiadis. 2025. RELINCHE: Automatically Checking Linearizability under Relaxed Memory Consistency. *Proc. ACM Program. Lang.* 9, POPL (2025), 2090–2117. https://doi.org/10.1145/3704906

Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 287–294. https://www.usenix.org/conference/atc22/presentation/gouk

Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. 2006. Open MPI: A flexible high performance MPI. In *Parallel Processing and Applied Mathematics: 6th International Conference, PPAM 2005, Poznań, Poland, September 11-14, 2005, Revised Selected Papers 6*. Springer, 228–239.

R. Gupta, V. Tipparaju, J. Nieplocha, and D. Panda. 2002. Efficient barrier using remote memory operations on VIA-based clusters. In *Proceedings. IEEE International Conference on Cluster Computing*. 83–90. https://doi.org/10.1109/CLUSTR.2002.1137732

Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*. Santorini, Greece, 355–364.

Maurice Herlihy and Jeannette M. Wing. 1990a. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

Maurice P. Herlihy and Jeannette M. Wing. 1990b. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (July 1990), 463–492.

George Hodgkins, Mark Madler, and Joseph Izraelevitz. 2025. LOCO: Rethinking Objects for Network Memory. (2025). arXiv:cs.DC/2503.19270 https://arxiv.org/abs/2503.19270

Joseph Izraelevitz, Gaukas Wang, Rhett Hanscom, Kayli Silvers, Tamara Silbergleit Lehman, Gregory Chockler, and Alexey Gotsman. 2023. Acuerdo: Fast Atomic Broadcast over RDMA. In *Proceedings of the 51st International Conference on Parallel Processing (ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 59, 11 pages. https://doi.org/10.1145/3545008.3545041

Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. 2019. Derecho: Fast State Machine Replication for Cloud Services. *ACM Trans. Comput. Syst.* 36, 2, Article 4 (April 2019), 49 pages. https://doi.org/10.1145/3302258

Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Mae Milano, Weijia Song, Edward Tremel, Robbert van Renesse, Sydney Zink, and Kenneth P. Birman. 2018. Derecho: Fast State Machine Replication for Cloud Services. *ACM Trans. Comput. Syst.* 36, 2 (2018), 4:1–4:49. https://doi.org/10.1145/3302258

Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Sydney Zink, Ken Birman, and Robbert Van Renesse. 2017. Building Smart Memories and High-speed Cloud Services for the Internet of Things with Derecho. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. ACM, New York, NY, USA, 632–632. https://doi.org/10.1145/3127479.3134597 Extended version available from www.cs.cornell.edu/ken/derecho-tocs.pdf.

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

Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 295–306. https://doi.org/10.1145/2619239.2626299

Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. 2015. Turning Centralized Coherence and Distributed Critical-Section Execution on their Head: A New Approach for Scalable Distributed Shared

Memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2749246.2749250

Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. 1994. TreadMarks: distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference (WTEC'94)*. USENIX Association, USA, 10.

Michalis Kokologiannakis and Viktor Vafeiadis. 2021. GenMC: A Model Checker for Weak Memory Models. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I (Lecture Notes in Computer Science)*, Alexandra Silva and K. Rustan M. Leino (Eds.), Vol. 12759. Springer, 427–440. https://doi.org/10.1007/978-3-030-81685-8_20

Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R Lebeck, and Danyang Zhuo. 2023. Understanding {RDMA} microarchitecture resources for performance isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 31–48.

Ori Lahav, Brijesh Dongol, and Heike Wehrheim. 2023. Rely-Guarantee Reasoning for Causally Consistent Shared Memory. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part I (Lecture Notes in Computer Science)*, Constantin Enea and Akash Lal (Eds.), Vol. 13964. Springer, 206–229. https://doi.org/10.1007/978-3-031-37706-8_11

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 618–632. https://doi.org/10.1145/3062341.3062352

Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 99–114. https://www.usenix.org/conference/fast23/presentation/li-pengfei

linux-rdma. 2018. RDMA core. (2018). https://github.com/linux-rdma/rdma-core/ (Accessed: Jul. 2025).

Feilong Liu, Claude Barthels, Spyros Blanas, Hideaki Kimura, and Garret Swart. 2021. Beyond MPI: New Communication Interfaces for Database Systems and Data-Intensive Applications. *SIGMOD Rec.* 49, 4 (March 2021), 12–17. https://doi.org/10.1145/3456859.3456862

Xu Liu and John Mellor-Crummey. 2014. A tool to analyze the performance of multithreaded programs on NUMA architectures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 259–272. https://doi.org/10.1145/2555243.2555271

Redis Ltd. 2021. Redis v6.0.16. (2021). https://github.com/redis/redis/releases/tag/6.0.16.

Redis Ltd. 2024. Memtier v2.1.2. (2024). https://github.com/RedisLabs/memtier_benchmark/releases/tag/2.1.2.

Kai Lu, Siqi Zhao, Haikang Shan, Qiang Wei, Guokuan Li, Jiguang Wan, Ting Yao, Huatao Wu, and Daohui Wang. 2024. Scythe: A Low-latency RDMA-enabled Distributed Transaction System for Disaggregated Memory. *ACM Trans. Archit. Code Optim.* 21, 3, Article 57 (Sept. 2024), 26 pages. https://doi.org/10.1145/3666004

Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. 2018. {Multi-Path} transport for {RDMA} in datacenters. In *15th USENIX symposium on networked systems design and implementation (NSDI 18)*. 357–371.

Zoltan Majo and Thomas R. Gross. 2017. A Library for Portable and Composable Data Locality Optimizations for NUMA Systems. *ACM Trans. Parallel Comput.* 3, 4, Article 20 (mar 2017), 32 pages. https://doi.org/10.1145/3040222

Message Passing Interface Forum. 2023. *MPI: A Message-Passing Interface Standard Version 4.1*. https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf

Adam Morrison and Yehuda Afek. 2013. Fast Concurrent Queues for x86 Processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 103–112. https://doi.org/10.1145/2442516.2442527

Ramana Nagasamudram, Lennart Beringer, Ken Birman, Mae Milano, and David A. Naumann. 2024. Verifying a C Implementation of Derecho's Coordination Mechanism Using VST and Coq. In *NASA Formal Methods - 16th International Symposium, NFM 2024, Moffett Field, CA, USA, June 4-6, 2024, Proceedings (Lecture Notes in Computer Science)*, Nathaniel Benz, Divya Gopinath, and Nija Shi (Eds.), Vol. 14627. Springer, 99–117. https://doi.org/10.1007/978-3-031-60698-4_6

J. Nieplocha, R.J. Harrison, and R.J. Littlefield. 1994. Global Arrays: a portable "shared-memory" programming model for distributed memory computers. In *Supercomputing '94:Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. 340–349. https://doi.org/10.1109/SUPERC.1994.344297

NVIDIA Corporation. 2020. NVIDIA Collective Communication Library (NCCL) Documentation. (2020). https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/index.html

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674.

Springer, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

Geoff Pike and Jyrki Alakuijala. [n. d.]. Introducing CityHash. ([n. d.]). https://opensource.googleblog.com/2011/04/introducing-cityhash.html.

Marius Poke and Torsten Hoefler. 2015. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*. ACM, New York, NY, USA, 107–118. https://doi.org/10.1145/2749246.2749267

Azalea Raad, Marko Doko, Lovro Rozic, Ori Lahav, and Viktor Vafeiadis. 2019. On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models. *Proc. ACM Program. Lang.* 3, POPL (2019), 68:1–68:31. https://doi.org/10.1145/3290381

Azalea Raad, Ori Lahav, John Wickerson, Piotr Balcer, and Brijesh Dongol. 2024. Intel PMDK Transactions: Specification, Validation and Concurrency. In *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II (Lecture Notes in Computer Science)*, Stephanie Weirich (Ed.), Vol. 14577. Springer, 150–179. https://doi.org/10.1007/978-3-031-57267-8_6

R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. 2007. *A Remote Direct Memory Access Protocol Specification.* Technical Report RFC 5040. Internet Engineering Task Force.

Yufei Ren, Xingbo Wu, Li Zhang, Yandong Wang, Wei Zhang, Zijun Wang, Michel Hack, and Song Jiang. 2017. iRDMA: Efficient Use of RDMA in Distributed Deep Learning Systems. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 231–238. https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.30

Zhenyuan Ruan, Shihang Li, Kaiyan Fan, Marcos K. Aguilera, Adam Belay, Seo Jin Park, and Malte Schwarzkopf. 2023a. Unleashing True Utility Computing with Quicksand. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HOTOS '23)*. Association for Computing Machinery, New York, NY, USA, 196–205. https://doi.org/10.1145/3593856.3595893

Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. 2023b. Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1409–1427. https://www.usenix.org/conference/nsdi23/presentation/ruan

Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan

Léo Stefanesco, Azalea Raad, and Viktor Vafeiadis. 2024. Specifying and Verifying Persistent Libraries. In *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II (Lecture Notes in Computer Science)*, Stephanie Weirich (Ed.), Vol. 14577. Springer, 185–211. https://doi.org/10.1007/978-3-031-57267-8_8

Lingjia Tang, Jason Mars, Xiao Zhang, Robert Hagmann, Robert Hundt, and Eric Tune. 2013. Optimizing Google's warehouse scale computers: The NUMA experience. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 188–197. https://doi.org/10.1109/HPCA.2013.6522318

Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A Memory-Disaggregated Managed Runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 261–280. https://www.usenix.org/conference/osdi20/presentation/wang

Jing Wang, Chao Li, Yibo Liu, Taolei Wang, Junyi Mei, Lu Zhang, Pengyu Wang, and Minyi Guo. 2023a. Fargraph+: Excavating the parallelism of graph processing workload on RDMA-based far memory system. *J. Parallel and Distrib. Comput.* 177 (2023), 144–159. https://doi.org/10.1016/j.jpdc.2023.02.015

Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1033–1048. https://doi.org/10.1145/3514221.3517824

Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, et al. 2023b. {SRNIC}: A scalable architecture for {RDMA}{NICs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1–14.

Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. 2019. Fast Distributed Deep Learning over RDMA. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 44, 14 pages. https://doi.org/10.1145/3302424.3303975

Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A Distributed File System for Non-Volatile Main Memories and RDMA-Capable Networks. In *17th USENIX Conference on File and Storage Technologies (FAST '19)*. USENIX Association.

Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2020. FileMR: Rethinking RDMA Networking for Scalable Persistent Memory. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation (NSDI'20)*. USENIX Association.

Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2022. Optimizing Data-intensive Systems in Disaggregated Data Centers with TELEPORT. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1345–1359. https://doi.org/10.1145/3514221.3517856

Yili Zheng, Amir Kamil, Michael B. Driscoll, Hongzhang Shan, and Katherine Yelick. 2014. UPC++: A PGAS Extension for C++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 1105–1114. https://doi.org/10.1109/IPDPS.2014.115

Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 523–536.

A Further Details of LOCO's Design

Our Library of Composable Objects (LOCO) is functionally an extension of the normal shared memory programming model, that is, an object-oriented paradigm, onto the weak memory network of RDMA. LOCO provides the ability to encapsulate network memory access within special objects, which we call *channels*. Channels are similar to traditional shared memory objects, in that they export methods, control their own memory and members, and manage their synchronisation. However, unlike traditional shared memory objects, a single channel may use memory across multiple nodes, including both network-accessible memory and private local memory. Examples of some channel types (classes) in LOCO include cross-node mutexes, barriers, queues, and maps.

A.1 Channel Overview

A LOCO application will usually consist of many channels (objects) of many different channel types (classes). In addition, each channel can itself instantiate member sub-channels (for instance, a key-value store might include several mutexes as sub-channels to synchronise access to its contents). We argue that such a system of channels makes it significantly easier to develop applications on network memory, without sacrificing performance.

Figure 18a shows our implementation of a barrier channel, based on [Gupta et al. 2002], using a SST sub-channel. As with a traditional shared memory barrier, it is used synchronise all participants at a certain point in execution. For each use of the barrier, participants increment their local, private, count variable, then broadcast the new value to others using their register in the SST. They then wait locally to leave the barrier until all participants have a count in the SST not less than their own.

A.2 Channel Setup

Figure 18b shows a complete example LOCO application: a microbenchmark which repeatedly waits on the barrier (Line 59) and measures its latency. At line 54, we construct the `manager` object from a set of (ID, hostname) pairs. The `manager` establishes connections with peers and mediates access to per-node resources: peer connections, a shared completion queue, and network-accessible memory.

The `manager` is then used to construct channel endpoints, in this case the barrier and its sub-channels (Line 55). Note that the barrier has a name "`bar`", which must match the name of the remote barrier endpoints to complete the connection. We use a '`/`' character to denote a sub-channel relationship (e.g., the full name of the SST in the barrier object is "`bar/sst`", with component `owned_var` s named "`bar/sst/ov0`" etc.), and a '`.`' character to denote a component memory region.

When a channel endpoint is constructed, it initialises its local state including subchannels, creates local memory regions, and indicates by name what memory regions it expects other participants to provide. Then, it sends a *join* message (Line 45) to each peer with the channel name and the list of memory regions it expects that peer to provide.

When a peer receives a join message, it first checks if a channel endpoint with the same name exists locally, and ignores the message if not (in other words, peers may not participate in all channels). If it finds a matching endpoint, it verifies its allocated memory regions match those requested, and returns a *connect* message containing metadata necessary to access the requested regions. Channels can also register callbacks which run when join and/or connect messages are received; these are used to create per-participant sub-channels or memory regions.

```
19 class barrier : public loco::channel {
20   unsigned count,num_nodes;
21   loco::sst_var<unsigned> sst;
22   public:
23   void waiting() {
24     // complete all outstanding RDMA
           operations
25     mgr()::fence();
26     count++; // increment our counter
27     sst.store_mine(count);
28     sst.push_broadcast(); //and push
29     bool waiting = true;
30     while(waiting){  // wait for others
31       waiting = false; // to match
32       for (auto& row : sst) {
33         if (row.load() < count){
34           waiting = true;
35           break;}
36       }
37     }
38   }
39   barrier(channel* parent,
40    string name,manager& cm,int num):
41    channel(parent, name, cm,
42    channel::expect_num(num-1)),
43    sst(this,"sst",cm)){
44     count=0; num_nodes=num;
45     channel::join();
46   }
47 };
```

(a) Complete C++ code for the network barrier, a simple channel object.

```
48 int main(int argc, char** argv) {
49   map<uint32_t, string> hosts;
50   int node_id, num_nodes;
51   loco::parse_hosts(&hosts,
52    &node_id,&num_nodes,argv[1]);
53   vector<timespec> lats;
54   loco::manager cm(ip_addrs, node_id);
55   loco::barrier bar("bar", cm,
         num_nodes);
56   cm.wait_for_ready();
57   for(int i=0; i<TEST_ITERS; ++i){
58     timespec t0 = clock_now();
59     bar.waiting();
60     timespec t1 = clock_now();
61     lats.push_back(t1 - t0);
62   }
63   cout<<"Avg_latency:"<<
64    accumulate(lats.begin(),
65    lats.end(),0.0))/lats.size();
66 }
```

(b) A simple (complete) LOCO application measuring barrier latency.

Fig. 18. LOCO barrier code

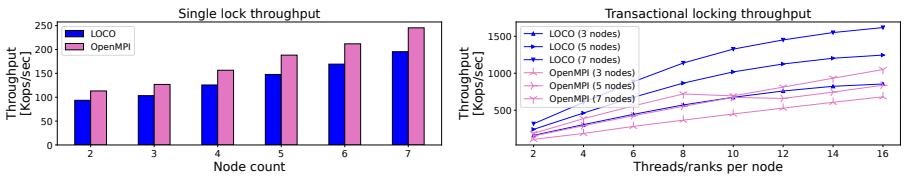


Fig. 19. Throughput of single-lock and transactions in OpenMPI and LOCO.

B Further LOCO-based Applications

B.1 Transactional Locking

In this section, we compare the performance of LOCO to the RDMA APIs provided by OpenMPI [Gabriel et al. 2004] on tasks involving contended synchronisation. We compare against OpenMPI version 5.0.5, using RoCE support provided by the PML/UCX backend. Results for both benchmarks are shown in Figure 19 (geomean of five 20-second runs).

First, we measured the throughput of a contended single-lock critical section (lock-protected read-modify-write) at different node counts, with one rank/thread per node. Here, OpenMPI has a consistent advantage, likely due to extensive optimisation and a more managed environment.

Then, we measured the throughput of a transactional critical section, which acquires the locks corresponding to two different accounts (array entries), and transfers a randomly generated amount between them. We use 100 million accounts. For intra-node scaling, LOCO creates multiple threads,

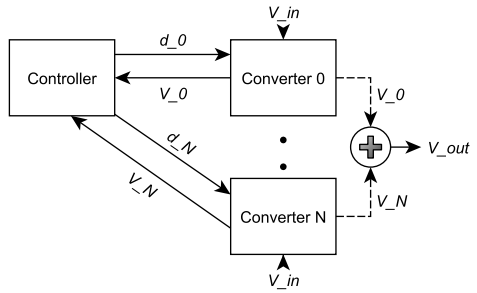


Fig. 20. Schematic of the system modeled by the power_controller channel. Solid arrows represent LOCO owned_vars, and dashed arrows represent electrical connections. d_N represents the duty cycle parameter used to control converter N, and V_N represents the output voltage at converter N.

while OpenMPI creates separate ranks (MPI processes), due to MPI's limited support for multi-threading within a rank.

For LOCO, we create an array of `atomic_vars` holding account values, striped across participants. For OpenMPI, we distribute the accounts across 341 windows (symmetrically allocated regions of remote memory, each associated with a single lock per rank); 341 is the maximum supported. To ensure a fair comparison, LOCO uses at most 341 locks per thread.

LOCO outperforms OpenMPI on transactional locking, despite the fact that we use an equal number of locks and their lock performs better in isolation. We believe this is due to the tight coupling between memory windows and locks in MPI: windows likely have a one-to-one correspondence with RDMA memory regions in the backend, and performing operations on many small memory regions is slower than large ones due to NIC caching structures [Kong et al. 2023]. LOCO avoids this penalty by disassociating regions and locks in its object system, while also merging regions into 1 GB huge pages in the backend.

B.2 Distributed DC/DC Converter System

As an additional application of LOCO, we implemented a model of a hardware control loop which exploits its low latency.

B.2.1 System Design. An additional application channel we have implemented is the `power_controller`, a real-time simulation of a distributed DC/DC converter system controlled by a discrete-time control loop [Corradini et al. 2015]. The simulation (Figure 20) consists of a single machine which acts as a *controller*, and an arbitrary number of machines simulating the physical characteristics of a *converter*. The role of the controller is to regulate the duty cycles (d) of the converters, which are supplied with a steady input DC voltage, to produce a target output voltage (V_{ref}). The converters return voltage values (V) which are used to calculate the next setting of their duty cycles, closing the control loop.

The `power_controller` channel consists of two arrays of `owned_vars` representing the duty cycle (owned by the controller) and output voltage (owned by the converter) for each converter. The participating machines run fixed-time loops: each loop iteration at a converter calculates a new simulated V and pushes it to the controller, while each iteration at the controller calculates a new d for all controllers based on their most recent d and V values. The overall output voltage of the system at each step (as seen by the controller) is the sum of all converters' most recent output voltage.

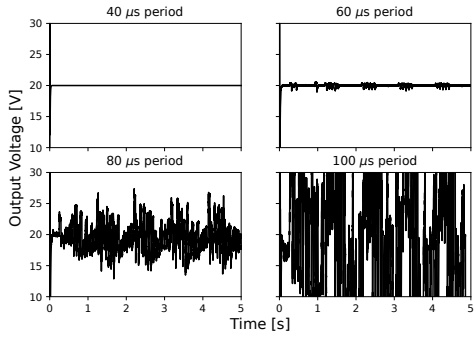


Fig. 21. Output voltage for the DC/DC converter simulation at various control loop frequencies.

Network memory is a good fit for this application because it is highly sensitive to network latency; with the parameters we have chosen, the output will only converge if latency of the control and feedback messages is consistently less than 40 μs. This requirement would be difficult to meet with traditional message-passing protocols: while a protocol such as UDP can easily achieve this latency on an uncontended network, it would be difficult to manage the scheduling jitter, copying, and cache contention in the software network protocol stack.

An extension of this control loop harness was developed in LOCO for validating hardware components such as the power controller and converters within partially simulated environments (hardware-in-loop testing). The system is currently in beta testing for production use, with expected commercial release later this year.

B.2.2 Evaluation. To evaluate whether LOCO meets the latency requirements of this system, we instantiated a cluster with one controller and 20 converters and measured the output voltage over time at various loop periods. The effect of changing the loop period is to simulate higher link latency, since we cannot increase the latency of the RDMA link. The loop period at the converters is fixed at 10 μs to approximate the continuous nature of their transfer function. We ran each simulation for 5 seconds.

The system parameters are selected to maintain a stable output voltage with a controller loop period of 40 μs or lower. The increasing instability in the output resulting from increasing the loop period past this value is clearly visible in Figure 21. The series with period greater than 40 μs also exhibit large transients at simulation start. These are mostly invisible on the plots due to their brief duration, but would be unacceptable in a real system.

C LOCO Backend

In this section, we briefly describe key features of the LOCO RDMA backend, which we have tuned extensively to expose the full performance of RDMA to LOCO applications (Section 6.2).

The LOCO backend uses the `libibverbs` library for RDMA communication, and the `librdmacm` library to manage RDMA connections. Both of these libraries are components of the Linux `rdma – core` project [rdm [n. d.]]. LOCO currently supports only RoCE [rdm 2014] as a link layer, although the only element missing for InfiniBand support is an implementation of the connection procedure. The current design assumes a reliable, static network of IP-addressable peers specified at application startup (the `hostnames` map declared at line 50 of Figure 18b).

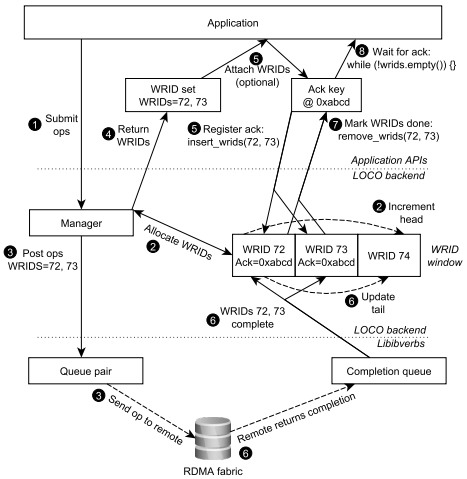


Fig. 22. Initiating and monitoring the progress of an RDMA operation in LOCO. If the application does not monitor the progress of the operation by registering the WRIDs returned in step 4 with an ack_key (step 5), then steps 7 and 8 do not occur. In addition, if the operation is already complete before the ack_key is registered (step 6 occurs before step 5), the WRIDs will not be inserted into the ack_key's internal set, and step 7 will not occur.

C.1 RDMA Operations and Completions

The most optimized portion of the backend is support for initiating RDMA operations and monitoring their progress, shown in Figure 22.

C.1.1 Initiating Operations. Applications submit operations to the backend using an `rdma_op` object which describes an operation, the relevant local memory, and one or more remote targets for the operation (in the case of writes; reads and atomics are semantically restricted to a single target). Encapsulating this information in a structure allows channels which are used to perform the same operation repeatedly (e.g. the rows of the SST used in the barrier) to memoize the operation description and simply pass the same object to the backend each time a new operation is initiated.

To reduce contention in multi-threaded applications, we create a new queue pair (QP) for each thread performing RDMA operations. These QPs are allocated "lazily," that is, they are created the first time that a thread calls into the backend to perform an operation. Using multiple QPs means that operations from different threads are not ordered with respect to each other, as described in Section C.2.

C.1.2 Monitoring Operation Progress. As mentioned above, Once an operation is posted to a QP, the backend returns a bitset to the user representing one or more Work Request IDs (WRIDs), 64-bit integers attached to individual RDMA operations to identify them throughout their lifetime. We use a single `libibverbs` completion queue (CQ), monitored by a dedicated *polling thread*, to track the status of outstanding operations.

WRIDs increase monotonically and are shared across the local node. The tail index of the queue tracks the oldest outstanding WRID, and the head tracks the most recently allocated WRID. WRIDs are mapped to slots in the WRID window by calculating `wrid%WINDOW_SIZE`. New WRIDs are allocated by performing an atomic fetch-and-add on the head index of the window. If some operations corresponding to the allocated slots are still outstanding from the previous "lap" (i.e., if $(\texttt{head} - \texttt{tail}) > \texttt{WINDOW\_SIZE}$), the allocator blocks until those operations complete. We found

that, as long as the window is sufficiently large (our evaluation uses 64K slots), this blocking only occurs if the thread count exceeds the hardware core count, likely due to the polling thread being descheduled by the OS.

The polling thread dequeues batches of completed operations from the CQ, and, after checking for errors, applies them to the WRID window. Each slot corresponding to a WRID in the batch is marked complete, as described below. In addition, if the batch contains the operation corresponding to the tail, the polling thread finds the new oldest outstanding operation, and updates the tail to point to that slot.

If the application wishes to monitor the progress of some RDMA operations, it creates an ack_key object and "attaches" it to the WRIDs for those operations. The ack_key contains a WRID bitset supporting atomic insertion and removal. When the ack_key is created, it accesses the WRID window to determine the status of each WRID in the input set. If the tail is greater than a given WRID, or if the slot for that WRID is already marked complete, the ack_key takes no action, avoiding any ordering requirement between an operation completing and registering it with an ack_key. Otherwise, the ack_key appends a pointer to itself to the queue slot, and inserts the WRID into its internal bitset.

When the polling thread marks a slot complete, it iterates over the attached list of ack_key s and clears the bit corresponding to that slot's WRID in each one's internal bitset. This is done atomically with respect to the registration process described above. Thus, on the application side, checking whether the operations attached to an ack_key are complete simply consists of checking whether the internal bitset is all zeroes, avoiding any explicit synchronization with the polling thread.

C.2 Fences and Memory Ordering

RDMA's specification of memory ordering is weak and unintuitive. To resolve these issues, we adopt the solution suggested by the specification: completion of a read operation guarantees that all earlier writes on that QP have been both completed and placed, meaning that any read acts as a global visibility fence for a given QP [Recio et al. 2007].

LOCO provides the option to add such a fence to any RDMA operation by appending a zero-length read, although it is only necessary when the application needs to wait until a write is globally visible, for instance when performing a write followed by a mutex release. If a fence is requested, it "takes over" the WRID of the fenced operation, and we do not request a completion for the fenced operation, making the fence transparent to the ack_key mechanism.

C.3 Local Scalability

LOCO implements multiple features aimed at increasing the scalability of performing RDMA operations across multiple local threads. First, each thread in a LOCO application uses a private set of Queue Pairs (one RDMA communication channel per peer), to avoid unnecessary synchronisation when multiple threads perform RDMA operations simultaneously. Second, all completions are delivered to a single completion queue, which is monitored by a dedicated *polling thread*, in order to avoid contention on the completion queue.

Application code can monitor the progress of one or more operations by registering an ack_key object (modelled as work identifiers "*d*") with the polling thread, which provides APIs for polling and waiting on completion of the operation. Internally, the ack_key is a lock-free bitset with bits mapped to in-progress operations. As operations complete, the polling thread clears the corresponding bits, so that checking for completion of an ack_key's registered operations simply consists of testing whether the internal bitset is equal to zero (i.e., empty). This approach avoids

explicit synchronisation between the polling thread and application threads waiting for operations to complete.

C.4 Network Memory Management

Another important service the backend provides is management of each node's network memory. Memory must be registered with `libibverbs` before it can be accessed remotely. Since registration of a memory region incurs non-negligible latency, we aggregate all registered memory used by LOCO channels into a series of 1GB huge pages, each of which corresponds to a single `libibverbs` memory region. The named memory region objects constructed by channels each correspond to a contiguous sub-range of one of these regions. Using huge pages reduces TLB utilisation, which can have a significant performance impact on multithreaded applications [Chen et al. 1992].

In addition to memory regions explicitly created by channels, we found it useful to create a primitive for allocating temporary chunks of network memory used as inputs and outputs of channel methods, which we call `mem_refs`. We allocate of backing memory for these objects from a per-thread pool of fixed-size block, which are in turn allocated from the larger pool of registered memory described above.

Finally, LOCO also provides the capacity to allocate local memory regions backed by *device memory*, which resides on the network card. RDMA accesses to device memory are faster than those to system memory, since they are not required to traverse the PCIe bus to main memory. However, since device memory is not coherent with main memory, it is mainly useful for holding state exclusively accessed through the network, such as mutex state.

D Clients using Multiple Libraries

Now that multiple libraries have been defined separately, we can write programs combining methods from all of them. The synchronisation guarantees of each library (i.e. so) are available to other libraries (via hb) to restrict the behaviours of the whole program. In the example of Fig. 23, a first thread sends data to a second using a ring buffer, and the two threads synchronise through a barrier, making sure the data is available. I.e., if the submit method succeeds, the receive method also has to succeed.

Ring Buffer x	Barrier z
$a := \mathsf{Submit}^{\mathrm{RBL}}(x, 1)$ $\mathsf{BAR}_{\mathrm{BAL}}(z)$	$\mathsf{BAR}_{\mathrm{BAL}}(z)$ $b := \mathsf{Receive}^{\mathrm{RBL}}(x)$
$(a, b) = (\mathtt{true}, \bot)$ ✗	$(a, b) = (\mathtt{true}, 1)$ ✓

Fig. 23. ring buffer + barrier example

As an illustration, let us show we cannot have a $\{\mathrm{RBL}, \mathrm{BAL}\}$-consistent execution $\mathcal{G} = \langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle$ corresponding to the disallowed behaviour $(a, b) = (\mathtt{true}, \bot)$. This result corresponds to E containing the four events

- $\mathrm{e}_S = \langle t_1, \_, \langle \mathsf{Submit}^{\mathrm{RBL}}, (x, 1), \mathtt{true}\rangle\rangle$,
- $\mathrm{e}_{B1} = \langle t_1, \_, \langle \mathsf{BAR}_{\mathrm{BAL}}, (z), ()\rangle\rangle$,
- $\mathrm{e}_{B2} = \langle t_2, \_, \langle \mathsf{BAR}_{\mathrm{BAL}}, (z), ()\rangle\rangle$, and
- $\mathrm{e}_R = \langle t_2, \_, \langle \mathsf{Receive}^{\mathrm{RBL}}, (x), \bot\rangle\rangle$

with $\mathrm{po} = \{\langle \mathrm{e}_S, \mathrm{e}_{B1}\rangle; \langle \mathrm{e}_{B2}, \mathrm{e}_R\rangle\}$. $\{\mathrm{RBL}, \mathrm{BAL}\}$-consistency (Def. 3.6) would imply:

(1) $(\mathrm{ppo} \cup \mathrm{so}|_{\mathrm{RBL}} \cup \mathrm{so}|_{\mathrm{BAL}})^+ \subseteq \mathrm{hb}$ is irreflexive;
(2) $\langle \{\mathrm{e}_S; \mathrm{e}_R\}, \emptyset, \mathtt{stmp}|_{\mathrm{RBL}}, \mathrm{so}|_{\mathrm{RBL}}, \_\rangle \in \mathrm{RBL}.\mathcal{C}$; and
(3) $\langle \{\mathrm{e}_{B1}; \mathrm{e}_{B2}\}, \emptyset, \mathtt{stmp}|_{\mathrm{BAL}}, \mathrm{so}|_{\mathrm{BAL}}, \_\rangle \in \mathrm{BAL}.\mathcal{C}$.

Assuming the consistency conditions of the two libraries, there is a single possibility for `stmp`:

- $\mathtt{stmp}(\mathrm{e}_S) = \{\mathsf{aCW}, \mathsf{aNRW}_{n_2}\}$;
- $\mathtt{stmp}(\mathrm{e}_{B1}) = \mathtt{stmp}(\mathrm{e}_{B2}) = \{\mathsf{aGF}_{n_1}, \mathsf{aGF}_{n_2}, \mathsf{aCR}\}$; and
- $\mathtt{stmp}(\mathrm{e}_R) = \{\mathsf{aWT}\}$.

For the barrier library, we necessarily have $c_z = 1$, $o(\mathsf{e}_{B1}) = o(\mathsf{e}_{B2}) = 1$, and thus the two events synchronise. Notably, we have $\langle \mathsf{e}_{B1}, \mathtt{aGF}_{n_2}\rangle \xrightarrow{\mathrm{so}|_{\mathrm{BAL}}} \langle \mathsf{e}_{B2}, \mathtt{aCR}\rangle$. For the ring buffer library, there is no succeeding receive and $\mathrm{rf} = \emptyset$. We thus have $\langle \mathsf{e}_R, \mathtt{aWT}\rangle \xrightarrow{\mathrm{fb}} \langle \mathsf{e}_S, \mathtt{aNRW}_{n_2}\rangle$, with $\mathrm{fb} = \mathrm{so}|_{\mathrm{RBL}}$. From the definition of to (Fig. 10), we have $\langle \mathsf{e}_S, \mathtt{aNRW}_{n_2}\rangle \xrightarrow{\mathrm{ppo}} \langle \mathsf{e}_{B1}, \mathtt{aGF}_{n_2}\rangle$ and $\langle \mathsf{e}_{B2}, \mathtt{aCR}\rangle \xrightarrow{\mathrm{ppo}} \langle \mathsf{e}_R, \mathtt{aWT}\rangle$. This implies an hb cycle between the four subevents, and the execution cannot be $\{\textsc{rbl}, \textsc{bal}\}$-consistent.

E Correctness Proof of the Mowgli Framework

As mentioned in the paper, for $f : A \to B$ and $r \subseteq A \times A$, we note $f(r) \triangleq \{\langle f(x), f(y)\rangle \mid \langle x, y\rangle \in r\}$. It is straightforward to show that $f(r_1 \cup r_2) = f(r_1) \cup f(r_2)$ and $f(r|_{A'}) \subseteq f(r)|_{f(A')}$ for any subset $A' \subseteq A$. When r is a strict partial order, we write $r|_{\mathrm{imm}}$ for the *immediate* edges in r, i.e. $r \setminus (r;r)$.

E.1 Wide Abstraction

First, we generalise the notion of abstractions (Def. 3.11) to *wide abstractions*.

Definition E.1. Suppose I is a well-defined implementation of a library L using Λ, and that $G = \langle E, \mathrm{po}\rangle$ and $G' = \langle E', \mathrm{po}'\rangle$ are plain executions using methods of Λ and $(\Lambda \uplus \{L\})$ respectively. We say that a function $f : E \to E'$ is a wide abstraction of G to G', denoted $\mathtt{wideabs}^f_{I,L}(G, G')$, iff

- $E' = f(E)$, i.e. $f : E \to E'$ is surjective;
- $E|_L = \emptyset$, i.e. G contains no calls to the abstract library L;
- $f(x) \notin L \implies f(x) = x$, i.e. events not part of an implementation of L are kept unchanged;
- $f(\mathrm{po}) \subseteq (\mathrm{po}')^*$ and $\forall \mathsf{e}_1, \mathsf{e}_2,\ \langle f(\mathsf{e}_1), f(\mathsf{e}_2)\rangle \in \mathrm{po}' \implies \langle \mathsf{e}_1, \mathsf{e}_2\rangle \in \mathrm{po}$; and
- if $\mathsf{e}' = \langle t, \iota, \langle m, \widetilde{v}, v'\rangle\rangle \in E'$ then $\langle\langle v', 0\rangle, G|_{f^{-1}(\mathsf{e}')}\rangle \in [\![I(t, m, \widetilde{v})]\!]_t$

The difference with the normal abstraction is that G' is not limited to methods of L, but every method call not from L is carried over to the implementation G (i.e. in general $E \cap E' \neq \emptyset$) and the abstraction function f maps these events to themselves.

E.2 Finding a Wide Abstraction

Lemma E.2. *Given $\widetilde{\mathsf{p}}$ and an implementation I of L using Λ, if $\langle \widetilde{v}, G\rangle \in [\![\lfloor\widetilde{\mathsf{p}}\rfloor_I]\!]$ then there is $\langle \widetilde{v}, G'\rangle \in [\![\widetilde{\mathsf{p}}]\!]$ and f such that $\mathtt{wideabs}^f_{I,L}(G, G')$.*

Proof. It is enough to show the following: for all t and p, if $\langle\langle v, k\rangle, \langle E, \mathrm{po}\rangle\rangle \in [\![\lfloor\mathsf{p}\rfloor_{t,I}]\!]_t$ then there is $\langle\langle v, k\rangle, \langle E', \mathrm{po}'\rangle\rangle \in [\![\mathsf{p}]\!]_t$ and f such that $\mathtt{wideabs}^f_{I,L}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$. Indeed, if this holds, we can conclude by merging the results of each thread for the case $k = 0$.

For a given t, we proceed by induction on p.

- If $\mathsf{p} = v$ or $\mathsf{p} = \mathtt{break}_{k'}\ v$, then $\lfloor\mathsf{p}\rfloor_{t,I} = \mathsf{p}$ and we have $\langle E, \mathrm{po}\rangle = \emptyset_G$. We simply take $\langle E, \mathrm{po}\rangle = \emptyset_G$ and we have $\mathtt{wideabs}^f_{I,L}(\emptyset_G, \emptyset_G)$ for the empty function f.
- If $\mathsf{p} = m(\widetilde{v_0})$ with $m \notin L.M$, then $\lfloor\mathsf{p}\rfloor_{t,I} = \mathsf{p}$ and we have $\langle E, \mathrm{po}\rangle = \{\mathsf{e}\}_G$ for some event e. We can choose $\langle E', \mathrm{po}'\rangle = \langle E, \mathrm{po}\rangle$. We have $\mathtt{wideabs}^{\mathtt{Id}}_{I,L}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$ for the identity function $\mathtt{Id}$ that maps e to itself:
  $\mathtt{Id}(\{\mathsf{e}\}) = \{\mathsf{e}\}$; $E|_L = \emptyset$; $\mathtt{Id}(\mathsf{e}) = \mathsf{e}$; $\mathrm{po}' = \emptyset = \mathrm{po}$; and the last property holds since $E'|_L = \emptyset$.
- If $\mathsf{p} = m(\widetilde{v_0})$ with $m \in L.M$, then $\lfloor\mathsf{p}\rfloor_{t,I} = I(t, m, \widetilde{v_0})$ and $\langle\langle v, k\rangle, \langle E, \mathrm{po}\rangle\rangle \in [\![I(t, m, \widetilde{v_0})]\!]_t$. By definition of the implementation, we have $k = 0$ and $E \neq \emptyset$. Let $\mathsf{e}' = (t, \iota, \langle m, \widetilde{v_0}, v\rangle)$ for some ident ι, we take $\langle E', \mathrm{po}'\rangle = \langle\{\mathsf{e}'\}, \emptyset\rangle$ and we indeed have $\langle\langle v, 0\rangle, \langle E', \mathrm{po}'\rangle\rangle \in [\![m(\widetilde{v_0})]\!]_t$. We choose the function f that maps every element of E to e', and we need to check $\mathtt{wideabs}^f_{I,L}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$.

We do have $\{e'\} = f(E)$ since $E \neq \emptyset$. We have $E|_L = \emptyset$ since I does not use L. Forall $x \in E$, $f(x) = e' \in L$. If $(e_1, e_2) \in \mathrm{po}$ then $f(e_1) = e' = f(e_2)$. $\mathrm{po}' = \emptyset$. Finally, for $e' \in \{e'\}$, we have $f^{-1}(e') = E$ and $\langle\langle v, 0\rangle, \langle E, \mathrm{po}\rangle\rangle \in [\![I(t, m, \widetilde{v_0})]\!]_t$ holds.

- If $\mathtt{p} = \mathtt{let}\ \mathtt{p}_1\ \mathtt{p}_2$, then $[\![\mathtt{let}\ \mathtt{p}_1\ \mathtt{p}_2]\!]_{t,I} \triangleq \mathtt{let}\ [\![\mathtt{p}_1]\!]_{t,I}\ (\lambda v.[\![\mathtt{p}_2\ v]\!]_{t,I})$, and $\langle\langle v, k\rangle, \langle E, \mathrm{po}\rangle\rangle \in [\![\mathtt{let}\ [\![\mathtt{p}_1]\!]_{t,I}\ (\lambda v.[\![\mathtt{p}_2\ v]\!]_{t,I})]\!]_t$ has two possible sources.
  - If $\langle\langle v, k\rangle, \langle E, \mathrm{po}\rangle\rangle \in [\![[\![\mathtt{p}_1]\!]_{t,I}]\!]_t$ (and $k \neq 0$), then by induction hypothesis we have E', po', and f such that $\langle\langle v, k\rangle, \langle E', \mathrm{po}'\rangle\rangle \in [\![\mathtt{p}_1]\!]_t$ and $\mathtt{wideabs}^f_{I,L}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$. Then $\langle\langle v, k\rangle, \langle E', \mathrm{po}'\rangle\rangle \in [\![\mathtt{p}]\!]_t$ also holds and we are done.
  - Else there is $E_1, E_2, \mathrm{po}_1, \mathrm{po}_2, v'$ such that $E = E_1 \cup E_2$, $\mathrm{po} = \mathrm{po}_1 \cup \mathrm{po}_2 \cup (E_1 \times E_2)$, $\langle\langle v', 0\rangle, \langle E_1, \mathrm{po}_1\rangle\rangle \in [\![[\![\mathtt{p}_1]\!]_{t,I}]\!]_t$, and $\langle\langle v, k\rangle, \langle E_2, \mathrm{po}_2\rangle\rangle \in [\![[\![\mathtt{p}_2\ v]\!]_{t,I}]\!]_t$. By induction hypothesis, there is $E'_1, E'_2, \mathrm{po}'_1, \mathrm{po}'_2, f_1, f_2$ such that $\langle\langle v', 0\rangle, \langle E'_1, \mathrm{po}'_1\rangle\rangle \in [\![\mathtt{p}_1]\!]_t$, $\langle\langle v, k\rangle, \langle E'_2, \mathrm{po}'_2\rangle\rangle \in [\![\mathtt{p}_2\ v]\!]_t$, $\mathtt{wideabs}^{f_1}_{I,L}(\langle E_1, \mathrm{po}_1\rangle, \langle E'_1, \mathrm{po}'_1\rangle)$, and $\mathtt{wideabs}^{f_2}_{I,L}(\langle E_2, \mathrm{po}_2\rangle, \langle E'_2, \mathrm{po}'_2\rangle)$. We choose $E' = E'_1 \cup E'_2$, $\mathrm{po}' = \mathrm{po}'_1 \cup \mathrm{po}'_2 \cup (E'_1 \times E'_2)$ and we have $\langle\langle v, k\rangle, \langle E', \mathrm{po}'\rangle\rangle \in [\![\mathtt{let}\ \mathtt{p}_1\ (\lambda v.\mathtt{p}_2\ v)]\!]_t$ by definition. We define $f : (E_1 \cup E_2) \to (E'_1 \cup E'_2)$ as the sum of f_1 (on E_1) and f_2 (on E_2). We are left to show $\mathtt{wideabs}^f_{I,L}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$.
    - $E' = E'_1 \cup E'_2 = f_1(E_1) \cup f_2(E_2) = f(E_1 \cup E_2) = f(E)$
    - $(E_1 \cup E_2)|_L = E_1|_L \cup E_2|_L = \emptyset$
    - If $x \in E_i$ and $f(x) = f_i(x) \notin L$, then $f(x) = x$ since the property holds for f_i
    - $f(\mathrm{po}) = f(\mathrm{po}_1 \cup \mathrm{po}_2 \cup (E_1 \times E_2)) = f(\mathrm{po}_1) \cup f(\mathrm{po}_2) \cup (f(E_1) \times f(E_2)) \subseteq \mathrm{po}'_1 \cup \mathtt{Id}|_{E'_1} \cup \mathrm{po}'_2|_{E'_2} \cup (E'_1 \times E'_2) = \mathrm{po}' \cup \mathtt{Id}$.
    - If $(f(e_1), f(e_2)) \in \mathrm{po}' = \mathrm{po}'_1 \cup \mathrm{po}'_2 \cup (E'_1 \times E'_2)$, then we have three cases. If $(f(e_1), f(e_2)) \in \mathrm{po}'_1$, then $(e_1, e_2) \in \mathrm{po}_1 \subseteq \mathrm{po}$. If $(f(e_1), f(e_2)) \in \mathrm{po}'_2$, then $(e_1, e_2) \in \mathrm{po}_2 \subseteq \mathrm{po}$. Finally, if $f(e_1) \in E'_1$ and $f(e_2) \in E'_2$, then $e_1 \in E_1$ and $e_2 \in E_2$, and so $(e_1, e_2) \in (E_1 \times E_2) \subseteq \mathrm{po}$.
    - If $e' = (t, \iota, \langle m, \widetilde{v}, v'\rangle) \in E'_i|_L$, from our hypothesis we know $\langle\langle v', 0\rangle, \langle E_i, \mathrm{po}_i\rangle|_{f_i^{-1}(e')}\rangle \in [\![I(t, m, \widetilde{v})]\!]_t$. We simply have $\langle E_i, \mathrm{po}_i\rangle|_{f_i^{-1}(e')} = \langle E, \mathrm{po}\rangle|_{E_i}|_{f_i^{-1}(e')} = \langle E, \mathrm{po}\rangle|_{f^{-1}(e')}$, so $\langle\langle v', 0\rangle, \langle E, \mathrm{po}\rangle|_{f^{-1}(e')}\rangle \in [\![I(t, m, \widetilde{v})]\!]_t$ holds
- Similarly for $\mathtt{p}$ of the shape $\mathtt{loop}\ \mathtt{p}'$.

From this, we can conclude the lemma. Given $\widetilde{\mathtt{p}}$ and an implementation I of L using Λ, if $\langle(v_1, \ldots, v_T), G\rangle \in [\![[\![\widetilde{\mathtt{p}}]\!]_I]\!]$ then by definition G is of the form $G = \|_{1 \leq t \leq T}\ G_t$ and $\forall 1 \leq t \leq T. \langle\langle v_t, 0\rangle, G_t\rangle \in [\![[\![\widetilde{\mathtt{p}}(t)]\!]_{t,I}]\!]_t$. Using the result above, we have $G'_1, \ldots, G'_T$ and $f_1, \ldots, f_T$ such that $\langle\langle v_t, 0\rangle, G'_t\rangle \in [\![\widetilde{\mathtt{p}}(t)]\!]_t$ and $\mathtt{wideabs}^{f_t}_{I,L}(G_t, G'_t)$. We define $G' = \|_{1 \leq t \leq T}\ G'_t$ and $f : G'.E \to G.E$ the sum of $f_1, \ldots, f_T$. We have $\langle\widetilde{v}, G'\rangle \in [\![\widetilde{\mathtt{p}}]\!]$ by definition, and we can easily show $\mathtt{wideabs}^f_{I,L}(G, G')$ similarly to the proof above. □

E.3 Locally Sound Implies Sound

Theorem E.3. *If a well-defined implementation is locally sound, then it is sound.*

Proof. Let I be a locally sound implementation of L using Λ. Let $\widetilde{\mathtt{p}}$ such that $\mathtt{loc}(I) \cap \mathtt{loc}(\widetilde{\mathtt{p}}) = \emptyset$. We need to show that $\mathtt{outcome}_\Lambda([\![\widetilde{\mathtt{p}}]\!]_I) \subseteq \mathtt{outcome}_{\Lambda \uplus \{L\}}(\widetilde{\mathtt{p}})$.

Let $\langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle$ Λ-consistent such that $\langle\widetilde{v}, \langle E, \mathrm{po}\rangle\rangle \in [\![[\![\widetilde{\mathtt{p}}]\!]_I]\!]$. From Lemma E.2, there is E', po', f such that $\langle\widetilde{v}, \langle E', \mathrm{po}'\rangle\rangle \in [\![\widetilde{\mathtt{p}}]\!]$ and $\mathtt{wideabs}^f_{I,L}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$.

Let $E_L \triangleq E'|_L$ and $E_p \triangleq E' \setminus E_L$. We also note $\mathrm{po}_L \triangleq \mathrm{po}'|_{E_L}$ and $\mathrm{po}_p \triangleq \mathrm{po}'|_{E'_p}$. By definition of $\mathtt{wideabs}^f_{I,L}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$, we have $E_p \subseteq E$ and $f|_{E_p} = \mathtt{Id}|_{E_p}$: by surjectivity if $e \in E_p$ then there is $e_0 \in E$ such that $f(e_o) = e$, but since $e \notin L$ we have $f(e_o) = e_o = e$ and thus $e \in E$.

We note $E_i = E \backslash E_p$, and create notations such that $\langle E_i, \mathrm{po}_i, \mathtt{stmp}_i, \mathrm{so}_i, \mathrm{hb}_i\rangle = \langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle|_{E_i}$ and $\langle E_p, \mathrm{po}_p, \mathtt{stmp}_p, \mathrm{so}_p, \mathrm{hb}_p\rangle = \langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle|_{E_p}$. Thus $E' = E_L \cup E_p$ and $E = E_i \cup E_p$. Intuitively, E_i is the implementation of E_L while the common part E_p is not modified.

We note $f_i = f|_{E_i}$. We can easily check that $\mathtt{abs}^{f_i}_{I,L}(\langle E_i, \mathrm{po}_i\rangle, \langle E_L, \mathrm{po}_L\rangle)$ holds:

- $E_L = f_i(E_i)$: Let $\mathrm{e} \in E_L$, since f is surjective there is $\mathrm{e}' \in E$ such that $f(\mathrm{e}') = \mathrm{e}$. Since $f|_{E_p} = \mathtt{Id}|_{E_p}$, for $\mathrm{e}_0 \in E_p$ we have $f(\mathrm{e}_0) = \mathrm{e}_0 \notin E_L$, so $\mathrm{e}' \in E_i$.
- $E_i|_L = \emptyset$ since $E_i \subseteq E$ and $E|_L = \emptyset$.
- $E_L = E_L|_L$ by definition.
- Let $\mathrm{e}_1, \mathrm{e}_2 \in E_i$, if $f_i(\mathrm{e}_1) \neq f_i(\mathrm{e}_2)$ then by $\mathtt{wideabs}^f_{I,L}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$ we have $(f_i(\mathrm{e}_1), f_i(\mathrm{e}_2)) \in \mathrm{po}'|_{E_L} = \mathrm{po}_L$.
- Let $\mathrm{e}_1, \mathrm{e}_2 \in E_i$ such that $(f_i(\mathrm{e}_1), f_i(\mathrm{e}_2)) \in \mathrm{po}_L \subseteq \mathrm{po}$. By $\mathtt{wideabs}^f_{I,L}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$ we have $(\mathrm{e}_1, \mathrm{e}_2) \in \mathrm{po}|_{E_i} = \mathrm{po}_i$.
- Let $\mathrm{e}' = (t, \iota, \langle m, \widetilde{v}, v'\rangle) \in E_L$. From $\mathtt{wideabs}^f_{I,L}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$ we have $\langle\langle v', 0\rangle, \langle E, \mathrm{po}\rangle|_{f^{-1}(\mathrm{e}')}\rangle \in [\![I(t, m, \widetilde{v})]\!]_t$. Since $\mathrm{e}' \in E_L$ and $f|_{E_p} = \mathtt{Id}|_{E_p}$, we have $f^{-1}(\mathrm{e}') = f_i^{-1}(\mathrm{e}') \subseteq E_i$ and thus $\langle E, \mathrm{po}\rangle|_{f^{-1}(\mathrm{e}')} = \langle E_i, \mathrm{po}_i\rangle|_{f_i^{-1}(\mathrm{e}')}$. So $\langle\langle v', 0\rangle, \langle E_i, \mathrm{po}_i\rangle|_{f_i^{-1}(\mathrm{e}')}\rangle \in [\![I(t, m, \widetilde{v})]\!]_t$.

Next, let us show that $\langle E_i, \mathrm{po}_i, \mathtt{stmp}_i, \mathrm{so}_i, \mathrm{hb}_i\rangle$ is Λ-consistent. The first two points are trivial, and we need to show that for any library $L' \in \Lambda$ we have $\langle E_i, \mathrm{po}_i, \mathtt{stmp}_i, \mathrm{so}_i, \mathrm{hb}_i\rangle|_{L'}$ L'-consistent. By hypothesis, we already know that $\langle (E_i \cup E_p), \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle|_{L'}$ is L'-consistent. Thus, by the decomposability property, it would be enough to show that $\mathtt{loc}(E_i) \cap \mathtt{loc}(E_p) = \emptyset$. Since $E_p \subseteq E'$ and $\langle \widetilde{v}, \langle E', \mathrm{po}'\rangle\rangle \in [\![\widetilde{\mathrm{p}}]\!]$, we know that $\mathtt{loc}(E_p) \subseteq \mathtt{loc}(E') \subseteq \mathtt{loc}(\widetilde{\mathrm{p}})$. Since $\mathtt{loc}(I) \cap \mathtt{loc}(\widetilde{\mathrm{p}}) = \emptyset$, it would be enough to show $\mathtt{loc}(E_i) \subseteq \mathtt{loc}(I)$. Let $\mathrm{e} \in E_i$, we have $f_i(\mathrm{e}) \in E_L$ of the form $(t, \iota, \langle m, \widetilde{v_0}, v'\rangle)$. By definition of local abstraction, $\langle\langle v', 0\rangle, \langle E_L, \mathrm{po}_L\rangle|_{f_i^{-1}(f_i(\mathrm{e}))}\rangle \in [\![I(t, m, \widetilde{v_0})]\!]_t$ and $\mathrm{e} \in E_L|_{f_i^{-1}(f_i(\mathrm{e}))}$. By definition of implementation, we have $\mathtt{loc}(\mathrm{e}) \subseteq \mathtt{loc}(E_L|_{f_i^{-1}(f_i(\mathrm{e}))}) \subseteq \mathtt{loc}(I)$.

Since I is locally sound, we can use $\langle E_i, \mathrm{po}_i, \mathtt{stmp}_i, \mathrm{so}_i, \mathrm{hb}_i\rangle$ Λ-consistent and $\mathtt{abs}^{f_i}_{I,L}(\langle E_i, \mathrm{po}_i\rangle, \langle E_L, \mathrm{po}_L\rangle)$ to produce $\mathtt{stmp}_L$, g_i, and so_L such that:

- $g_i(\mathrm{e}', a') = (\mathrm{e}, a)$ implies $f_i(\mathrm{e}) = \mathrm{e}'$ and
  - Forall a_0 such that $(a_0, a') \in \mathrm{to}$, there exists $(\mathrm{e}_1, a_1) \in \mathrm{SEvent}_i$ such that $f_i(\mathrm{e}_1) = \mathrm{e}'$, $(a_0, a_1) \in \mathrm{to}$, and $((\mathrm{e}_1, a_1), (\mathrm{e}, a)) \in (\mathrm{hb}_i \cup \mathtt{Id})$;
  - Forall a_0 such that $(a', a_0) \in \mathrm{to}$, there exists $(\mathrm{e}_2, a_2) \in \mathrm{SEvent}_i$ such that $f_i(\mathrm{e}_2) = \mathrm{e}'$, $(a_2, a_0) \in \mathrm{to}$, and $((\mathrm{e}, a), (\mathrm{e}_2, a_2)) \in (\mathrm{hb}_i \cup \mathtt{Id})$.
- $g_i(\mathrm{so}_L) \subseteq \mathrm{hb}_i$;
- Forall hb_L transitive such that $(\mathrm{ppo}_L \cup \mathrm{so}_L)^+ \subseteq \mathrm{hb}_L$ and $g_i(\mathrm{hb}_L) \subseteq \mathrm{hb}_i$, we have $\langle E_L, \mathrm{po}_L, \mathtt{stmp}_L, \mathrm{so}_L, \mathrm{hb}_L\rangle \in L.C$, where $\mathrm{ppo}_L \triangleq \langle E_L, \mathrm{po}_L, \mathtt{stmp}_L\rangle.\mathrm{ppo}$.

We define $\mathtt{stmp}'$ on E' by the sum of $\mathtt{stmp}_L$ and $\mathtt{stmp}_p$. We define $\mathrm{so}' \triangleq \mathrm{so}_L \cup \mathrm{so}_p$, as well as $\mathrm{ppo}' \triangleq \langle E', \mathrm{po}', \mathtt{stmp}'\rangle.\mathrm{ppo}$, and $\mathrm{hb}' \triangleq (\mathrm{ppo}' \cup \mathrm{so}')^+$. We extend $g_i : \langle E_L, \mathtt{stmp}_L\rangle.\mathrm{SEvent} \to \langle E_i, \mathtt{stmp}_i\rangle.\mathrm{SEvent}$ into a function $g : \langle E', \mathrm{po}', \mathtt{stmp}'\rangle.\mathrm{SEvent} \to \langle E, \mathrm{po}, \mathtt{stmp}\rangle.\mathrm{SEvent}$ using the identity function. I.e., for $(\mathrm{e}', a') \in \langle E_p, \mathtt{stmp}_p\rangle.\mathrm{SEvent}$ we have $g(\mathrm{e}', a') = (\mathrm{e}', a')$. The first property above on g_i and f_i carries over to g and f, by taking, for any stamp, the intermediary subevents to be the output of g itself.

As an important intermediary result, let us show $g(\mathrm{hb}') \subseteq \mathrm{hb}$. Since $\mathrm{hb}' = (\mathrm{ppo}' \cup \mathrm{so}')^+$, we need to show the inclusion for each component. $g(\mathrm{so}') = g_i(\mathrm{so}_L) \cup \mathtt{Id}(\mathrm{so}_p) \subseteq \mathrm{hb}_i \cup \mathrm{so}_p \subseteq \mathrm{hb}$ is immediate, and we are left with proving $g(\mathrm{ppo}') \subseteq \mathrm{hb}$. Let $(\mathrm{e}'_1, a'_1), (\mathrm{e}'_2, a'_2) \in \mathrm{SEvent}'$ such that $(\mathrm{e}'_1, \mathrm{e}'_2) \in \mathrm{po}'$ and $(a'_1, a'_2) \in \mathrm{to}$. Let $(\mathrm{e}_i, a_i) \triangleq g(\mathrm{e}'_i, a'_i)$ $(i \in \{1, 2\})$, we need to show that $((\mathrm{e}_1, a_1), (\mathrm{e}_2, a_2)) \in \mathrm{hb}$. From the properties of g, using (e_1, a_1) and the stamp a_2, there is (e^g_1, a^g_1) such that $f(\mathrm{e}^g_1) = \mathrm{e}'_1$, $(a^g_1, a_2) \in \mathrm{to}$, and $((\mathrm{e}_1, a_1), (\mathrm{e}^g_1, a^g_1)) \in \mathrm{hb}$. From the properties on g, using (e_2, a_2) and the stamp

a_1^g, there is (e_2^g, a_2^g) such that $f(e_2^g) = e_2'$, $(a_1^g, a_2^g) \in \mathsf{to}$, and $((e_2^g, a_2^g), (e_2, a_2)) \in \mathsf{hb}$. We know that $e_1' = f(e_1^g)$ and $e_2' = f(e_2^g)$, so from $\mathtt{wideabs}^f_{I,L}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$ and $(e_1', e_2') \in \mathrm{po}'$ we have $(e_1^g, e_2^g) \in \mathrm{po}$. Thus $((e_1^g, a_1^g), (e_2^g, a_2^g)) \in \mathrm{ppo}$, and by transitivity we have $((e_1, a_1), (e_2, a_2)) \in \mathsf{hb}$. This finishes the intermediary result $g(\mathsf{hb}') \subseteq \mathsf{hb}$.

To conclude the theorem, we want to show that $\langle E', \mathrm{po}', \mathtt{stmp}', \mathrm{so}', \mathsf{hb}'\rangle$ is $(\Lambda \cup \{L\})$-consistent.

- The first few points hold because hb' is irreflexive, since $g(\mathsf{hb}') \subseteq \mathsf{hb}$ and hb is irreflexive.
- For $L' \in \Lambda$, we need to show that $\langle E_p, \mathrm{po}_p, \mathtt{stmp}_p, \mathrm{so}_p, \mathsf{hb}'\rangle|_{L'}$ is L'-consistent. We know that $\langle E_p, \mathrm{po}_p, \mathtt{stmp}_p, \mathrm{so}_p, \mathsf{hb}_p\rangle|_{L'}$ is L'-consistent, so by monotonicity it is enough to show $\mathsf{hb}'|_{E_p} \subseteq \mathsf{hb}|_{E_p}$, which holds because $\mathsf{hb}'|_{E_p} = g(\mathsf{hb}')|_{E_p} \subseteq \mathsf{hb}|_{E_p}$.
- For L, we need to show that $\langle E_L, \mathrm{po}_L, \mathtt{stmp}_L, \mathrm{so}_L, \mathsf{hb}'|_L\rangle$ is L-consistent. Using our hypothesis, it is enough to show that $\mathsf{hb}_L \triangleq \mathsf{hb}'|_L$, which is transitive and includes $(\mathrm{ppo}_L \cup \mathrm{so}_L)^+$, satisfies $g_i(\mathsf{hb}_L) \subseteq \mathsf{hb}_i$. Once again, this holds because $g_i(\mathsf{hb}_L) = g(\mathsf{hb}'|_{E_L}) \subseteq g(\mathsf{hb}')|_{E_i} \subseteq \mathsf{hb}|_{E_i} = \mathsf{hb}_i$.

□

F RDMA$^{\text{WAIT}}$ implementation into RDMA$^{\text{TSO}}$

F.1 Background: RDMA$^{\text{TSO}}$

Our definition of RDMA$^{\text{TSO}}$ is closer to an independent language than a library. Unlike the definition of an execution in Definition 3.3, we do not need a relation hb to represent the potential rest of the program, as RDMA$^{\text{TSO}}$ is not a library in the sense of Definition 3.5. A program *cannot* combine instructions from RDMA$^{\text{TSO}}$ and other libraries presented in this paper, as polling would interfere with RDMA operations of other libraries.

We use the following 11 methods:

$$\begin{aligned} m(\overline{v}) ::= {} & \mathtt{Write}^{\text{TSO}}(x, v) \mid \mathtt{Read}^{\text{TSO}}(x) \mid \mathtt{CAS}^{\text{TSO}}(x, v_1, v_2) \mid \mathtt{Mfence}^{\text{TSO}}() \\ & \mid \mathtt{Get}^{\text{TSO}}(x, y) \mid \mathtt{Put}^{\text{TSO}}(x, y) \mid \mathtt{Poll}(n) \mid \mathtt{Rfence}^{\text{TSO}}(n) \\ & \mid \mathtt{SetAdd}(x, v) \mid \mathtt{SetRemove}(x, v) \mid \mathtt{SetIsEmpty}(x) \end{aligned}$$

- $\mathtt{Write}^{\text{TSO}} : \mathsf{Loc} \times \mathsf{Val} \to ()$
- $\mathtt{Read}^{\text{TSO}} : \mathsf{Loc} \to \mathsf{Val}$
- $\mathtt{CAS}^{\text{TSO}} : \mathsf{Loc} \times \mathsf{Val} \times \mathsf{Val} \to \mathsf{Val}$
- $\mathtt{Mfence}^{\text{TSO}} : () \to ()$
- $\mathtt{Get}^{\text{TSO}} : \mathsf{Loc} \times \mathsf{Loc} \to \mathsf{Val}$
- $\mathtt{Put}^{\text{TSO}} : \mathsf{Loc} \times \mathsf{Loc} \to \mathsf{Val}$
- $\mathtt{Poll} : \mathsf{Node} \to \mathsf{Val}$
- $\mathtt{Rfence}^{\text{TSO}} : \mathsf{Node} \to ()$
- $\mathtt{SetAdd} : \mathsf{Loc} \times \mathsf{Val} \to ()$
- $\mathtt{SetRemove} : \mathsf{Loc} \times \mathsf{Val} \to ()$
- $\mathtt{SetIsEmpty} : \mathsf{Loc} \to \mathbb{B}$

As expected, the `Wait` operation is replaced with a `Poll` operation. Compared to RDMA$^{\text{TSO}}$ from [Ambal et al. 2024], we slightly extend the language so that put/get operations return an arbitrary unique identifier, and polling also returns the same identifier of the operation being polled[4]. In addition, we also assume basic set operations `SetAdd`, `SetRemove`, and `SetIsEmpty` to store these new identifiers, where the locations used for sets do not overlap with locations used for other operations.

Consistency predicate. An execution of an RDMA$^{\text{TSO}}$ program is of the form $\mathcal{G} = \langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}\rangle$, similarly to Def. 3.5 but $\mathsf{hb} = (\mathrm{ppo} \cup \mathrm{so})^+$ does not have the flexibility of containing additional external constraints.

We define the only valid stamping function $\mathtt{stmp}_{\text{TSO}}$ as follows:

[4]In practice, the identifier is not random and can be chosen by the program

- A poll has stamp aWT: $\mathtt{stmp}_{\mathsf{TSO}}((\_,\_,(\mathtt{Poll},\_,\_))) = \{\mathtt{aWT}\}$.
- Auxiliary set operations have stamp aMF: $\mathtt{stmp}_{\mathsf{TSO}}((\_,\_,(\mathtt{SetAdd},\_,\_))) = \mathtt{stmp}_{\mathsf{TSO}}((\_,\_,(\mathtt{SetRemove},\_,\_))) = \mathtt{stmp}_{\mathsf{TSO}}((\_,\_,(\mathtt{SetIsEmpty},\_,\_))) = \{\mathtt{aMF}\}$.
- Other events follow $\mathtt{stmp}_{\mathsf{RL}}$ (*cf.* Appendix F.2). E.g., events calling $\mathtt{Write}^{\mathrm{TSO}}$ have stamp aCW, while events calling $\mathtt{Get}^{\mathrm{TSO}}$ towards node n have stamps $\mathtt{aNRR}_n$ and $\mathtt{aNLW}_n$. We also define loc on subevents similarly to $\textsc{rdma}^{\mathrm{WAIT}}$.

We mark set operations with aMF to simplify the consistency conditions, as we do not want to explicitly integrate them in the read ($\mathcal{R}$) and write ($\mathcal{W}$) subevents.

Given $\mathcal{G} = \langle E, \mathrm{po}, \mathtt{stmp}_{\mathsf{TSO}}, \mathrm{so}\rangle$, we say that v_{R}, v_{W}, rf, mo, nfo, and pf are well-formed if:

- v_{R}, v_{W}, rf, mo, and nfo are well-formed, as in $\textsc{rdma}^{\mathrm{WAIT}}$;
- Let $P_n \triangleq \{(\mathsf{e}, \mathtt{aWT}) \mid \mathsf{e} = (\_,\_,(\mathtt{Poll},(n),\_)) \in E\}$ be the set of poll (sub)events towards node n. Then $\mathrm{pf} \subseteq \bigcup_{n\in\mathrm{Node}}(\mathcal{G}.\mathtt{aNLW}_n \cup \mathcal{G}.\mathtt{aNRW}_n) \times P_n$ is the *polls-from* relation, relating earlier NIC writes to later polls. Moreover:
  - $\mathrm{pf} \subseteq \mathrm{po}$ (we can only poll previous operations of the same thread);
  - pf is functional on its domain (every NIC write can be polled at most once);
  - pf is total and functional on its range (every Poll polls from exactly one NIC write);
  - Poll events poll-from the oldest non-polled remote operation towards the given node:
    for each node n, if $w_1, w_2 \in (\mathcal{G}.\mathtt{aNLW}_n \cup \mathcal{G}.\mathtt{aNRW}_n)$ and $w_1 \xrightarrow{\mathrm{po}} w_2 \xrightarrow{\mathrm{pf}} p_2$, then there exists p_1 such that $w_1 \xrightarrow{\mathrm{pf}} p_1 \xrightarrow{\mathrm{po}} p_2$;
  - and a Poll returns the unique identifier of the polled operation:
    if $((\_,\_,(\_,\_,v_1)),\_) \xrightarrow{\mathrm{pf}} ((\_,\_,(\mathtt{Poll},\_,v_2)),\mathtt{aWT})$ then $v_1 = v_2$.

We use the derived relations rb, $\mathrm{rb_i}$, $\mathrm{rf_e}$, $\mathrm{rf_i}$, ippo, and iso as defined for $\textsc{rdma}^{\mathrm{WAIT}}$. We can then define ib as follows:

$$\mathrm{ib} \triangleq (\mathrm{ippo} \cup \mathrm{iso} \cup \mathrm{rf} \cup \mathrm{pf} \cup \mathrm{nfo} \cup \mathrm{rb_i})^+$$

Definition F.1 ($\textsc{rdma}^{\mathrm{TSO}}$-consistency). $\mathcal{G} = \langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}\rangle$ is $\textsc{rdma}^{\mathrm{TSO}}$-consistent if:

- $(\mathrm{ppo} \cup \mathrm{so})^+$ is irreflexive (similarly to Definition 3.6);
- $\langle E, \mathrm{po}\rangle$ respects nodes (as in $\textsc{rdma}^{\mathrm{WAIT}}$);
- $\mathtt{stmp} = \mathtt{stmp}_{\mathsf{TSO}}$;
- there exists well-formed v_{R}, v_{W}, rf, mo, nfo, and pf such that ib is irreflexive and $\mathrm{so} = \mathrm{iso} \cup \mathrm{rf_e} \cup [\mathtt{aNLW}]; \mathrm{pf} \cup \mathrm{nfo} \cup \mathrm{rb} \cup \mathrm{mo} \cup ([\mathtt{Inst}]; \mathrm{ib})$;
- identifiers for get/put operations are unique:
  if e_1 and e_2 are both of the form $(\_,\_,(\mathtt{Get}^{\mathrm{TSO}},\_,v))$ or $(\_,\_,(\mathtt{Put}^{\mathrm{TSO}},\_,v))$, then $\mathsf{e}_1 = \mathsf{e}_2$;
- and the set operations are (per-thread) sound: if SetIsEmpty returns true, then every value added to the set was subsequently removed. I.e., if $\mathsf{e}_1 = (t,\_,(\mathtt{SetAdd},(x,v),\_))$, $\mathsf{e}_3 = (t,\_,(\mathtt{SetIsEmpty},(x),\mathtt{true}))$, and $\mathsf{e}_1 \xrightarrow{\mathrm{po}} \mathsf{e}_3$, then there exists $\mathsf{e}_2 = (t,\_,(\mathtt{SetRemove},(x,v),\_))$ such that $\mathsf{e}_1 \xrightarrow{\mathrm{po}} \mathsf{e}_2 \xrightarrow{\mathrm{po}} \mathsf{e}_3$.

F.2 $\textsc{rdma}^{\mathrm{WAIT}}$ Library

This appendix completes Section 3.3 on the definition of $\textsc{rdma}^{\mathrm{WAIT}}$. As mentioned, we have the 8 methods:

$$\begin{aligned} m(\widetilde{v}) ::= {} & \mathtt{Write}(x,v) \mid \mathtt{Read}(x) \mid \mathtt{CAS}(x,v_1,v_2) \mid \mathtt{Mfence}() \\ & \mid \mathtt{Get}(x,y,d) \mid \mathtt{Put}(x,y,d) \mid \mathtt{Wait}(d) \mid \mathtt{Rfence}(n) \end{aligned}$$

- $\texttt{Write} : \text{Loc} \times \text{Val} \to ()$
- $\texttt{Read} : \text{Loc} \to \text{Val}$
- $\texttt{CAS} : \text{Loc} \times \text{Val} \times \text{Val} \to \text{Val}$
- $\texttt{Mfence} : () \to ()$
- $\texttt{Get} : \text{Loc} \times \text{Loc} \times \text{Wid} \to ()$
- $\texttt{Put} : \text{Loc} \times \text{Loc} \times \text{Wid} \to ()$
- $\texttt{Wait} : \text{Wid} \to ()$
- $\texttt{Rfence} : \text{Node} \to ()$

We also define loc as expected: $\texttt{loc}(\texttt{Write}(x, v)) = \texttt{loc}(\texttt{Read}(x)) = \texttt{loc}(\texttt{CAS}(x, v_1, v_2)) = \{x\}$; $\texttt{loc}(\texttt{Get}(x, y, d)) = \texttt{loc}(\texttt{Put}(x, y, d)) = \{x; y\}$; and $\texttt{loc}(\texttt{e}) = \emptyset$ otherwise.

We assume that each location x is associated with a specific node $\texttt{n}(x)$. We say that $\langle E, \text{po}\rangle$ respects nodes if for all event on thread t with label of the form $(\texttt{Write}, (x, \_), \_)$, $(\texttt{Read}, (x), \_)$, $(\texttt{CAS}, (x, \_, \_), \_)$, $(\texttt{Get}, (x, \_, \_), \_)$, or $(\texttt{Put}, (\_, x, \_), \_)$, we have $\texttt{n}(x) = \texttt{n}(t)$. I.e. arguments corresponding to local locations should be locations of the current node. Given $\langle E, \text{po}\rangle$, we now define the only valid stamping function $\texttt{stmp}_{\text{RL}}$. Since the thread is not relevant, we note $\texttt{stmp}_{\text{RL}}(m(\widetilde{v}), v')$ for $\texttt{stmp}_{\text{RL}}(\langle \_, \_, \langle m, \widetilde{v}, v'\rangle\rangle)$.

- $\texttt{stmp}_{\text{RL}}(\texttt{Write}(x, v), ()) = \{\texttt{aCW}\}$
- $\texttt{stmp}_{\text{RL}}(\texttt{Read}(x), v) = \{\texttt{aCR}\}$
- $\texttt{stmp}_{\text{RL}}(\texttt{Mfence}(), ()) = \{\texttt{aMF}\}$
- $\texttt{stmp}_{\text{RL}}(\texttt{CAS}(x, v_1, v_2), v_1) = \{\texttt{aCAS}\}$
- $\texttt{stmp}_{\text{RL}}(\texttt{CAS}(x, v_1, v_2), v_3) = \{\texttt{aMF}; \texttt{aCR}\}$ if $v_1 \neq v_3$
- $\texttt{stmp}_{\text{RL}}(\texttt{Wait}(d), ()) = \{\texttt{aWT}\}$
- $\texttt{stmp}_{\text{RL}}(\texttt{Get}(x, y, d), ()) = \{\texttt{aNRR}_{\texttt{n}(y)}; \texttt{aNLW}_{\texttt{n}(y)}\}$
- $\texttt{stmp}_{\text{RL}}(\texttt{Put}(x, y, d), ()) = \{\texttt{aNLR}_{\texttt{n}(x)}; \texttt{aNRW}_{\texttt{n}(x)}\}$
- $\texttt{stmp}_{\text{RL}}(\texttt{Rfence}(n), ()) = \{\texttt{aRF}_n\}$

Put and get operations perform both a NIC read and a NIC write, and as such are associated to two stamps. A succeeding CAS can be represented as a single stamp aCAS, while a failing CAS behaves as both a memory fence (aMF) and a CPU read (aCR).

We extend loc to subevents. For events with zero or one locations, the subevents have the same set of locations. For Get/Put, each of the two subevent is associated to the relevant location. E.g. if $\texttt{e} = (\_, \_, (\texttt{Get}, (x, y, d), \_))$, then $\texttt{loc}(\langle \texttt{e}, \texttt{aNRR}_{\texttt{n}(y)}\rangle) = \{y\}$ and $\texttt{loc}(\langle \texttt{e}, \texttt{aNLW}_{\texttt{n}(y)}\rangle) = \{x\}$.

Given an execution $\mathcal{G} = \langle E, \text{po}, \texttt{stmp}_{\text{RL}}, \_, \_\rangle$, recall we define the set of *reads* as $\mathcal{G}.\mathcal{R} \triangleq \mathcal{G}.\texttt{aCR} \cup \mathcal{G}.\texttt{aCAS} \cup \mathcal{G}.\texttt{aNLR} \cup \mathcal{G}.\texttt{aNRR}$ and *writes* as $\mathcal{G}.\mathcal{W} \triangleq \mathcal{G}.\texttt{aCW} \cup \mathcal{G}.\texttt{aCAS} \cup \mathcal{G}.\texttt{aNLW} \cup \mathcal{G}.\texttt{aNRW}$. We say that $\texttt{v}_\texttt{R}$, $\texttt{v}_\texttt{W}$, rf, mo, and nfo are well-formed if:

- $\texttt{v}_\texttt{R} : \mathcal{G}.\mathcal{R} \to \text{Val}$ associates each read subevent with a value, matching the value returned if available: if e has a label of the form $(\texttt{Read}, \_, v)$ or $(\texttt{CAS}, \_, v)$, then $\texttt{v}_\texttt{R}(\texttt{e}) = v$.
- $\texttt{v}_\texttt{W} : \mathcal{G}.\mathcal{W} \to \text{Val}$ associates each write subevent with a value, matching the value written if known in $\mathcal{G}$: if e has a label of the form $(\texttt{Write}, (\_, v), \_)$ or $(\texttt{CAS}, (\_, v', v), v')$, then $\texttt{v}_\texttt{W}(\texttt{e}) = v$.
- RDMA operations write the value read: if $\texttt{s}_1 = \langle \texttt{e}, \texttt{aNLR}_n\rangle \in E$ and $\texttt{s}_2 = \langle \texttt{e}, \texttt{aNRW}_n\rangle \in E$, then $\texttt{v}_\texttt{R}(\texttt{s}_1) = \texttt{v}_\texttt{W}(\texttt{s}_2)$; and similarly for $\texttt{aNRR}_n$ and $\texttt{aNLW}_n$.
- $\text{rf} \subseteq \mathcal{G}.\mathcal{W} \times \mathcal{G}.\mathcal{R}$ is the '*reads-from*' relation on events of the same location with matching values; i.e. $(\texttt{s}_1, \texttt{s}_2) \in \text{rf} \implies \texttt{loc}(\texttt{s}_1) = \texttt{loc}(\texttt{s}_2) \wedge \texttt{v}_\texttt{W}(\texttt{s}_1) = \texttt{v}_\texttt{R}(\texttt{s}_2)$. rf is functional on its range: every read in $\mathcal{G}.\mathcal{R}$ is related to at most one write in $\mathcal{G}.\mathcal{W}$. If a read is not related to a write, it reads the initial value of zero: $\texttt{s}_2 \in \mathcal{G}.\mathcal{R} \wedge (\_, \texttt{s}_2) \notin \text{rf} \implies \texttt{v}_\texttt{R}(\texttt{s}_2) = 0$.
- $\text{mo} \triangleq \bigcup_{x \in \text{Loc}} \text{mo}_x$ is the '*modification-order*', where each mo_x is a strict total order on $\mathcal{G}.\mathcal{W}_x$ describing the order in which writes on x reach the memory.
- nfo is the '*NIC flush order*', such that for all n and $(\texttt{s}_1, \texttt{s}_2) \in \mathcal{G}.\text{SEvent}$ with $\texttt{t}(\texttt{s}_1) = \texttt{t}(\texttt{s}_2)$, if $(\texttt{s}_1, \texttt{s}_2) \in \mathcal{G}.\texttt{aNLR}_n \times \mathcal{G}.\texttt{aNLW}_n$ then $(\texttt{s}_1, \texttt{s}_2) \in \text{nfo} \cup \text{nfo}^{-1}$, and if $(\texttt{s}_1, \texttt{s}_2) \in \mathcal{G}.\texttt{aNRR}_n \times \mathcal{G}.\texttt{aNRW}_n$ then $(\texttt{s}_1, \texttt{s}_2) \in \text{nfo} \cup \text{nfo}^{-1}$.

The definitions above are similar to the relations defined for sv (see §3.4), with the addition of nfo representing the PCIe guarantees that NIC reads flush previous NIC writes.

For each subevent, we distinguish the moment the subevent *starts* executing and the moment it *finishes* executing. The relation so represents dependency between the end of executions of subevents. To express the semantics of $\text{RDMA}^{\text{WAIT}}$, we also need to consider the *issued-before* relation

ib representing dependency between the start of executions of subevents. Note that neither ib or so is a subset of the other. The starting (when sent to the store buffer of PCIe fabric) and finishing (reaching memory) points of some write subevents might differ. We define the set of *instantaneous subevents* as $\mathcal{G}.\texttt{Inst} \triangleq \mathcal{G}.\text{SEvent} \setminus (\mathcal{G}.\texttt{aCW} \cup \mathcal{G}.\texttt{aNLW} \cup \mathcal{G}.\texttt{aNRW})$, regrouping the subevents that start and finish at the same time.

Given $\mathcal{G}$ and well-formed $\mathsf{v_R}$, $\mathsf{v_W}$, rf, mo, and nfo, we derive additional relations.

$$\text{rb} \triangleq \left\{(r,w) \;\middle|\; \begin{array}{c} r \in \mathcal{G}.\mathcal{R} \wedge w \in \mathcal{G}.\mathcal{W} \wedge \texttt{loc}(r) = \texttt{loc}(w) \\ \wedge\ ((r,w) \in (\text{rf}^{-1}; \text{mo}) \vee r \notin \texttt{img}(\text{rf})) \end{array}\right\} \setminus [\mathcal{G}.\text{SEvent}] \qquad \text{rf}_\text{e} \triangleq \text{rf} \setminus \text{rf}_\text{i}$$

$$\text{pfg} \triangleq \left\{((\mathsf{e}_1, \texttt{aNLW}_n), (\mathsf{e}_2, \texttt{aWT})) \;\middle|\; \begin{array}{c} \exists d.\ (\mathsf{e}_1, \mathsf{e}_2) \in \text{po} \\ \wedge\ \mathsf{e}_1 = (\_, \_, (\texttt{Get}, (\_, \_, d), \_)) \\ \wedge\ \mathsf{e}_2 = (\_, \_, (\texttt{Wait}, (d), \_)) \end{array}\right\} \qquad \text{rf}_\text{i} \triangleq [\texttt{aCW}]; (\text{po} \cap \text{rf}); [\texttt{aCR}]$$

$$\text{pfp} \triangleq \left\{((\mathsf{e}_1, \texttt{aNRW}_n), (\mathsf{e}_2, \texttt{aWT})) \;\middle|\; \begin{array}{c} \exists d.\ (\mathsf{e}_1, \mathsf{e}_2) \in \text{po} \\ \wedge\ \mathsf{e}_1 = (\_, \_, (\texttt{Put}, (\_, \_, d), \_)) \\ \wedge\ \mathsf{e}_2 = (\_, \_, (\texttt{Wait}, (d), \_)) \end{array}\right\}$$

$$\text{rb}_\text{i} \triangleq [\texttt{aCR}]; ((\text{po} \cup \text{po}^{-1}) \cap \text{rb}); [\texttt{aCW}]$$

$$\begin{aligned} \text{iso} \triangleq\ & \{((\mathsf{e}, \texttt{aMF}), (\mathsf{e}, \texttt{aCR})) \mid \mathsf{e} = (\_, \_, (\texttt{CAS}, \_, \_)) \in E \wedge \texttt{stmp}_{\mathsf{RL}}(\mathsf{e}) = \{\texttt{aMF}; \texttt{aCR}\}\} \\ & \cup \{((\mathsf{e}, \texttt{aNRR}_n), (\mathsf{e}, \texttt{aNLW}_n)) \mid \mathsf{e} = (\_, \_, (\texttt{Get}, \_, \_)) \in E \wedge \texttt{stmp}_{\mathsf{RL}}(\mathsf{e}) = \{\texttt{aNRR}_n; \texttt{aNLW}_n\}\} \\ & \cup \{((\mathsf{e}, \texttt{aNLR}_n), (\mathsf{e}, \texttt{aNRW}_n)) \mid \mathsf{e} = (\_, \_, (\texttt{Put}, \_, \_)) \in E \wedge \texttt{stmp}_{\mathsf{RL}}(\mathsf{e}) = \{\texttt{aNLR}_n; \texttt{aNRW}_n\}\} \end{aligned}$$

pfg (resp pfp) represent the synchronisation between the write part of a get (resp put) and a later Wait on the same work identifier. While both are included in ib, only pfg is included in so as waiting for a put does not guarantee the NIC remote write has finished. We define rf_e, rf_i, and rb similarly to the semantics of sv, and we also define rb_i as expected. The internal synchronisation order iso represents ordering between subevents of the same event. We ask that puts and gets read before writing, and that a failing CAS performs a memory fence before reading.

Finally we can define ib as follows. ib includes a larger subset of po than ppo, as we guarantee the starting order of the cases corresponding to cells B1, B5, G10, and I10 of Fig. 10. I.e., while a later CPU read might finish before an earlier CPU write, they have to start in order; and while a remote fence does not guarantee previous NIC writes have finished, it guarantees they have at least started.

$$\text{ippo} \triangleq \text{ppo} \cup [\mathcal{G}.\texttt{aCW}]; \text{po}; [\mathcal{G}.\texttt{aCR} \cup \mathcal{G}.\texttt{aWT}] \cup \bigcup_{n \in \text{Node}} ([\mathcal{G}.\texttt{aNRW}_n \cup \mathcal{G}.\texttt{aNLW}_n]; \text{po}; [\mathcal{G}.\texttt{aRF}_n])$$

$$\text{ib} \triangleq (\text{ippo} \cup \text{iso} \cup \text{rf} \cup \text{pfg} \cup \text{pfp} \cup \text{nfo} \cup \text{rb}_\text{i})^+$$

And from this we define the consistency predicate for $\text{RDMA}^{\text{WAIT}}$, similarly to the semantics of RDMA^{TSO}. We ask that ib and so be irreflexive, the second being implied by Def. 3.6. The inclusion of ([Inst]; ib) in so indicates that, if an instantaneous subevent starts before another subevent, then they also finish in the same order.

Definition F.2 ($\text{RDMA}^{\text{WAIT}}$-consistency). $\mathcal{G} = \langle E, \text{po}, \texttt{stmp}, \text{so}, \text{hb}\rangle$ is $\text{RDMA}^{\text{WAIT}}$-consistent if:

- $\langle E, \text{po}\rangle$ respects nodes;
- $\texttt{stmp} = \texttt{stmp}_{\mathsf{RL}}$;

For a thread t using work identifiers $\{d_1, \ldots, d_K\}$:

$I_W(t, \texttt{Write}, (x, v)) \triangleq \texttt{Write}^{\text{TSO}}(x, v)$

$I_W(t, \texttt{Read}, (x)) \triangleq \texttt{Read}^{\text{TSO}}(x)$

$I_W(t, \texttt{CAS}, (x, v_1, v_2)) \triangleq \texttt{CAS}^{\text{TSO}}(x, v_1, v_2)$

$I_W(t, \texttt{Mfence}, ()) \triangleq \texttt{Mfence}^{\text{TSO}}()$

$I_W(t, \texttt{Rfence}, (n)) \triangleq \texttt{Rfence}^{\text{TSO}}(n)$

$I_W(t, \texttt{Get}, (x, y, d)) \triangleq$
$\texttt{let}\ v = \texttt{Get}^{\text{TSO}}(x, y)\ \texttt{in}\ \texttt{SetAdd}(d^{n(y)}, v)$

$I_W(t, \texttt{Wait}, (d)) \triangleq$
For n in $1, \ldots, N$ do {
  While (SetIsEmpty(d^n) $\neq$ true) do {
    let v = Poll(n) in
    For k in $1, \ldots, K$ do {
      SetRemove(d_k^n, v) } } }

$I_W(t, \texttt{Put}, (x, y, d)) \triangleq$
$\texttt{let}\ v = \texttt{Put}^{\text{TSO}}(x, y)\ \texttt{in}\ \texttt{SetAdd}(d^{n(x)}, v)$

Fig. 24. Implementation I_W of RDMA$^{\text{WAIT}}$ into RDMA$^{\text{TSO}}$

- there exists well-formed $\mathsf{v_R}$, $\mathsf{v_W}$, rf, mo, and nfo such that ib is irreflexive and so = iso ∪ rf_e ∪ pfg ∪ nfo ∪ rb ∪ mo ∪ ([Inst]; ib).

We can easily check that this predicate satisfies monotonicity and decomposability.

F.3 Implementation Function

In Fig. 24 we define the implementation I_W from a full program using only the RDMA$^{\text{WAIT}}$ library into a program using only RDMA$^{\text{TSO}}$. We assume threads use disjoint work identifiers $d \in \text{Wid}$, otherwise it is straightforward to rename them.

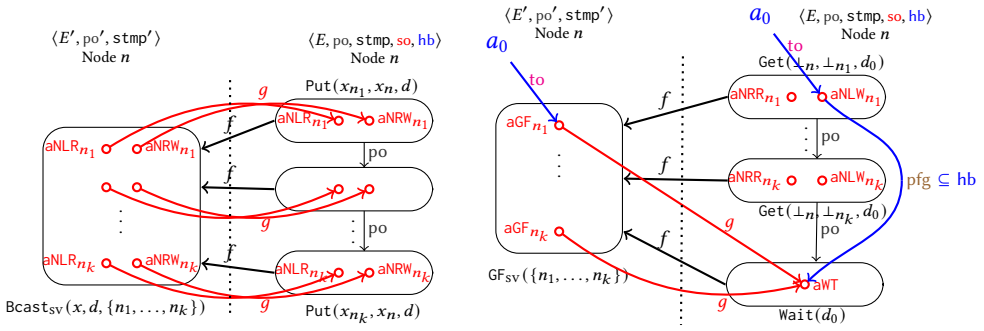
For each location x of RDMA$^{\text{WAIT}}$, we also use a location x for RDMA$^{\text{TSO}}$. For each work identifier d of RDMA$^{\text{WAIT}}$, we use new RDMA$^{\text{TSO}}$ locations $\{d^1, \ldots, d^N\}$ where $N \triangleq \#(\text{Node})$ is the number of nodes. Each location d^n is used as a set containing the identifiers of ongoing operations towards node n.

Most RDMA$^{\text{WAIT}}$ operations (Write, Read, CAS, Mfence, and Rfence) are directly translated into their RDMA$^{\text{TSO}}$ counterparts. An operation $\texttt{Get}(x, y, d)$ towards node n is translated into a similar $\texttt{Get}^{\text{TSO}}(x, y)$ whose output is added to the set d^n; We proceed similarly for puts. Finally, a $\texttt{Wait}(d)$ operation needs to poll until all relevant operations are finished, i.e. the sets $\{d^1, \ldots, d^N\}$ are all empty. Whenever we poll, we obtain the identifier of a finished operation, and we remove it from *all* sets where it might be held. We remove it from d^n but also from any other set d_k^n tracking a different group of operations, as otherwise a later call to $\texttt{Wait}(d_k)$ would hang and never return.

To simplify the notation of the implementation, we use the intuitive for-loops and while-loops. As no information is carried between the loops, these for-loops can be inlined, and the while-loops can easily be turned into loop-break similarly to Fig. 11.

F.4 Proof

We do not prove that the implementation above is locally sound (Definition 3.13), as Theorem 3.14 does *not* apply in this case. It is not possible to combine a program following RDMA$^{\text{TSO}}$ with programs of the other libraries presented in this paper. Instead, we assume a full program using only the RDMA$^{\text{WAIT}}$ library and compile it into RDMA$^{\text{TSO}}$.



(a) Concretisation function for broadcasts. (b) Concretisation function for global fence.

Fig. 25. Local soundness proof of I_{SV}: concretisation function g for broadcast and global fence, with $\mathsf{n}(t) = n$. For broadcast, each subevent of the sv program is mapped to a subevent of the implementation that uses the same stamp. For global fence, subevents with stamp $\mathsf{aGF}_{n'}$ are mapped to subevents with weaker stamps aWT, and we show that for any previous stamp a_0 a happens-before order can be given in the implementation (shown for the first subevent aGF_{n_1}).

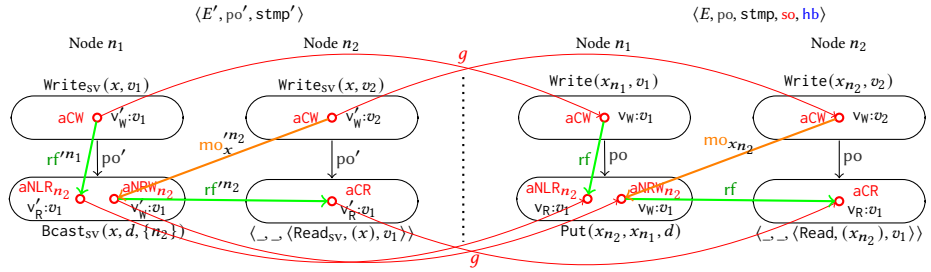


Fig. 26. Local soundness proof of I_{SV}: example of v'_{R}, v'_{W}, rf′, and mo′, defined from $\langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle$. These graphs correspond to the execution of $\mathtt{Write}_{\mathsf{sv}}(x, v_1); \mathtt{Bcast}_{\mathsf{sv}}(x, d, \{n_2\}) \parallel \mathtt{Write}_{\mathsf{sv}}(x, v_2); \mathtt{Read}_{\mathsf{sv}}(x)$ where the second node reads the value of the first node.

Theorem F.3. *Let $\widetilde{\mathrm{p}}$ be a program using only the RDMA^{WAIT} library. Then we have* $\mathtt{outcome}_{RDMA^{TSO}}(\llbracket\widetilde{\mathrm{p}}\rrbracket_{I_{\mathsf{W}}}) \subseteq \mathtt{outcome}_{\{RDMA^{WAIT}\}}(\widetilde{\mathrm{p}})$, *where:*

$$\mathtt{outcome}_{\{RDMA^{WAIT}\}}(\widetilde{\mathrm{p}}) = \left\{\widetilde{v} \;\middle|\; \exists\langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle\ \{RDMA^{WAIT}\}\text{-consistent}.\ \langle\widetilde{v}, \langle E, \mathrm{po}\rangle\rangle \in \llbracket\widetilde{\mathrm{p}}\rrbracket\right\}$$

$$\mathtt{outcome}_{RDMA^{TSO}}(\llbracket\widetilde{\mathrm{p}}\rrbracket_{I_{\mathsf{W}}}) = \left\{\widetilde{v} \;\middle|\; \exists\langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}\rangle\ RDMA^{TSO}\text{-consistent}.\ \langle\widetilde{v}, \langle E, \mathrm{po}\rangle\rangle \in \llbracket\llbracket\widetilde{\mathrm{p}}\rrbracket_{I_{\mathsf{W}}}\rrbracket\right\}$$

Proof. See Theorem G.9. □

G Correctness Proofs of the Core LOCO Libraries

G.1 sv Library

Theorem G.1. *The implementation I_{SV} of the sv library into RDMA^{WAIT} given in the paper is locally sound.*

Proof. We assume an $\{\text{RDMA}^{\text{WAIT}}\}$-consistent execution $\mathcal{G} = \langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle$ which is abstracted via f to $\langle E', \mathrm{po}'\rangle$ that uses the sv library, i.e. $\mathsf{abs}^{f}_{I_{\mathsf{SV}},\mathsf{SV}}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$ holds.

As a reminder, $\langle E', \mathrm{po}'\rangle$ (and later notations with an apostrophe) represents a client program using *only* calls to the sv library, while $\langle E, \mathrm{po}\rangle$ is the implementation of this program (via I_{SV}), thus

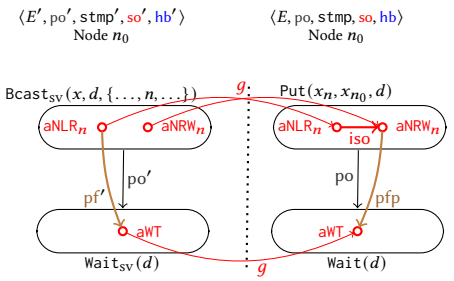


Fig. 27. Local soundness proof of I_{SV}: $g(\mathsf{pf}') \subseteq \mathsf{iso}; \mathsf{pfp} \subseteq ([\mathtt{Inst}]; \mathsf{ib}); \mathsf{ib} \subseteq \mathsf{hb}$.

using calls to the RDMA$^{\text{WAIT}}$ library. We assume the execution $\mathcal{G}$ of the implementation is consistent (w.r.t. RDMA$^{\text{WAIT}}$) and want to show the corresponding behaviour of the client program is consistent (w.r.t. the specification of SV in Definition 3.8), following the notion of local soundness (Def. 3.13).

Following Definition 3.13, we need to provide $\mathtt{stmp}'$, so', and $g : \langle E', \mathrm{po}', \mathtt{stmp}'\rangle.\mathrm{SEvent} \to \mathcal{G}.\mathrm{SEvent}$ respecting some conditions. From $\langle E', \mathrm{po}'\rangle$, we simply take $\mathtt{stmp}' = \mathtt{stmp}_{\mathsf{SV}}$. We note SEvent' for $\langle E', \mathrm{po}', \mathtt{stmp}'\rangle.\mathrm{SEvent}$.

Since $\mathcal{G}$ is $\{\text{RDMA}^{\text{WAIT}}\}$-consistent, it means $(\mathsf{ppo} \cup \mathsf{so})^+ \subseteq \mathsf{hb}$, hb is transitive and irreflexive, and $\mathcal{G}$ is RDMA$^{\text{WAIT}}$-consistent. Firstly, it means that for all thread t we have $\mathrm{po}|_t$ is a strict total order. From the properties of $\mathsf{abs}^f_{I_{\mathsf{SV}},\mathsf{SV}}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$, we can easily see that it implies $\mathrm{po}'|_t$ is also a strict total order. Secondly, there exists well-formed $\mathsf{v_R}$, $\mathsf{v_W}$, rf, mo, and nfo such that ib is irreflexive, $\mathtt{stmp} = \mathtt{stmp}_{\mathsf{RL}}$, and $\mathsf{so} = \mathsf{iso} \cup \mathsf{rf_e} \cup \mathsf{pfg} \cup \mathsf{nfo} \cup \mathsf{rb} \cup \mathsf{mo} \cup ([\mathtt{Inst}]; \mathsf{ib})$.

We define the concretisation function g as follows. The two involved cases are illustrated in Fig. 25.

- For an event $\mathsf{e}' = (t, \_, (\mathsf{Read}_{\mathsf{sv}}, (x), v))$, the only subevent is $(\mathsf{e}', \mathsf{aCR}) \in \mathrm{SEvent}'$. By definition of the abstraction f, the set $[\![I_{\mathsf{SV}}(t, \mathsf{Read}_{\mathsf{sv}}, (x))]\!]_t = [\![\mathsf{Read}(x_{\mathsf{n}(t)})]\!]_t = \{\langle\langle v', 0\rangle, \{(t, \iota, \langle \mathsf{Read}, x_{\mathsf{n}(t)}, v'\rangle)\}_G\rangle \mid v' \in \mathsf{Val} \wedge \iota \in \mathsf{ActionId}\}$ contains $\langle\langle v, 0\rangle, \langle E, \mathrm{po}\rangle|_{f^{-1}(\mathsf{e}')}\rangle$, so there is an event $\mathsf{e} = (t, \_, (\mathsf{Read}, (x_{\mathsf{n}(t)}), v)) \in E$ with $f(\mathsf{e}) = \mathsf{e}'$. From the definition of $\mathtt{stmp}_{\mathsf{RL}}$, it is associated to a single subevent $(\mathsf{e}, \mathsf{aCR}) \in \mathcal{G}.\mathrm{SEvent}$, and we define $g(\mathsf{e}', \mathsf{aCR}) = (\mathsf{e}, \mathsf{aCR})$. The first condition of g trivially holds for this input since the output uses the same stamp: for any stamp a_0 we can choose $(\mathsf{e}_1, a_1) = (\mathsf{e}_2, a_2) = (\mathsf{e}, \mathsf{aCR})$ using the $\mathtt{Id}$ function, and the to order is preserved for any previous or later stamp.
- For an event $\mathsf{e}' = (t, \_, (\mathsf{Write}_{\mathsf{sv}}, (x, v), ()))$, a similar reasoning allows us to choose $g(\mathsf{e}', \mathsf{aCW}) = (\mathsf{e}, \mathsf{aCW})$ with $\mathsf{e} = (t, \_, (\mathsf{Write}, (x_{\mathsf{n}(t)}, v), ())) \in f^{-1}(\mathsf{e}')$.
- For an event $\mathsf{e}' = (t, \_, (\mathsf{Wait}_{\mathsf{sv}}, (d), ()))$, a similar reasoning allows us to choose $g(\mathsf{e}', \mathsf{aWT}) = (\mathsf{e}, \mathsf{aWT})$ with $\mathsf{e} = (t, \_, (\mathsf{Wait}, (d), ())) \in f^{-1}(\mathsf{e}')$.
- For an event $\mathsf{e}' = (t, \_, (\mathsf{Bcast}_{\mathsf{sv}}, (x, d, \{n_1; \ldots; n_k\}), ()))$ and a subevent $(\mathsf{e}', \mathsf{aNLR}_{n_i})$, with $1 \le i \le k$, since the implementation of e' contains $\mathsf{Put}(x_{n_i}, x_{\mathsf{n}(t)}, d)$, the abstraction f similarly implies an event $\mathsf{e} = (t, \_, (\mathsf{Put}, (x_{n_i}, x_{\mathsf{n}(t)}, d), ())) \in f^{-1}(\mathsf{e}')$. This case is illustrated in Fig. 25a. As before, given $\mathtt{stmp}_{\mathsf{RL}}$, we can choose $g(\mathsf{e}', \mathsf{aNLR}_{n_i}) = (\mathsf{e}, \mathsf{aNLR}_{n_i})$ and the first condition on g holds using the identity function.
- Similarly for an event $\mathsf{e}' = (t, \_, (\mathsf{Bcast}_{\mathsf{sv}}, (x, d, \{n_1; \ldots; n_k\}), ()))$ and a subevent $(\mathsf{e}', \mathsf{aNRW}_{n_i})$, we can choose $g(\mathsf{e}', \mathsf{aNRW}_{n_i}) = (\mathsf{e}, \mathsf{aNRW}_{n_i})$ with $\mathsf{e} = (t, \_, (\mathsf{Put}, (x_{n_i}, x_{\mathsf{n}(t)}, d), ())) \in f^{-1}(\mathsf{e}')$. This is also illustrated in Fig. 25a.
- For an event $\mathsf{e}' = (t, \_, (\mathsf{GF}_{\mathsf{sv}}, (\{n_1; \ldots; n_k\}), ()))$ and a subevent $(\mathsf{e}', \mathsf{aGF}_{n_i})$, the relevant part of the implementation $f^{-1}(\mathsf{e}')$ of e' contains an event of label $\mathsf{Get}(\bot_{\mathsf{n}(t)}, \bot_{n_i}, d_0)$ (with stamps aNRR_{n_i} and aNLW_{n_i}) followed by one of label $\mathsf{Wait}(d_0)$ (with stamp aWT). See Fig. 25b for an illustration. Since the restrictive stamp aGF_{n_i} needs to be implemented using weaker

stamps, the choice of g is more delicate. We choose for g to map to the last subevent, i.e. $g(\mathsf{e}', \mathtt{aGF}_{n_i}) = (\mathsf{e}, \mathtt{aWT})$ with $\mathsf{e} = (t, \_, (\mathtt{Wait}, (d_0), ())) \in f^{-1}(\mathsf{e}')$, and we need to check the stamp ordering is preserved. For a later stamp a_0 such that $(\mathtt{aGF}_{n_i}, a_0) \in \mathsf{to}$, we can simply use $(\mathsf{e}_2, a_2) = (\mathsf{e}, \mathtt{aWT})$ using the `Id` function. We have $(\mathtt{aWT}, a_0) \in \mathsf{to}$ by definition (in Fig. 10, lines E and K are identical). For an earlier stamp a_0 such that $(a_0, \mathtt{aGF}_{n_i}) \in \mathsf{to}$, we use the entry point $(\mathsf{e}_1, a_1) = ((t, \_, (\mathtt{Get}, (\bot_{\mathsf{n}(t)}, \bot_{n_i}, d_0), ())), \mathtt{aNLW}_{n_i})$, as illustrated in Fig. 25b. As previously, we clearly have $\mathsf{e}_1 \in f^{-1}(\mathsf{e}')$. We have $(a_0, \mathtt{aNLW}_{n_i}) \in \mathsf{to}$ by definition (in Fig. 10, columns 9 and 11 are identical). We also need to check that $((\mathsf{e}_1, \mathtt{aNLW}_{n_i}), (\mathsf{e}, \mathtt{aWT})) \in \mathsf{hb}$. Since $\mathcal{G}$ is $\{\text{RDMA}^{\text{WAIT}}\}$-consistent, this is simply because $((\mathsf{e}_1, \mathtt{aNLW}_{n_i}), (\mathsf{e}, \mathtt{aWT})) \in \mathsf{pfg} \subseteq \mathsf{so} \subseteq \mathsf{hb}$ as the two events are in po and use the same identifier d_0.

Now we need to find so' such that $g(\mathsf{so}') \subseteq \mathsf{hb}$ and such that $\mathcal{G}' = \langle E', \mathsf{po}', \mathtt{stmp}', \mathsf{so}', \mathsf{hb}' \rangle$ is sv-consistent for any reasonable hb'. Actually, since hb' does not appear in the consistency predicate, we can ignore the properties of hb' and we need to check that $\langle E', \mathsf{po}', \mathtt{stmp}', \mathsf{so}', \_ \rangle$ is sv-consistent. For this, we need to choose well-formed v'_{R}, v'_{W}, rf', and mo'.

For v'_{R} and v'_{W}, we simply take $\mathsf{v}'_{\mathsf{R}}(\mathsf{s}') \triangleq \mathsf{v}_{\mathsf{R}}(g(\mathsf{s}'))$ and $\mathsf{v}'_{\mathsf{W}}(\mathsf{s}') \triangleq \mathsf{v}_{\mathsf{W}}(g(\mathsf{s}'))$. For the methods $\mathtt{Write}_{\mathsf{sv}}$ and $\mathtt{Read}_{\mathsf{sv}}$, these new functions v'_{R} and v'_{W} respect the value read/written, since v_{R} and v_{W} do so in $\mathcal{G}.\mathsf{SEvent}$. Similarly, if $\mathsf{s}'_1 = (\mathsf{e}', \mathtt{aNLR}_n) \in \mathsf{SEvent}'$ and $\mathsf{s}'_2 = (\mathsf{e}', \mathtt{aNRW}_n) \in \mathsf{SEvent}'$ (so e' calls the $\mathtt{Bcast}_{\mathsf{sv}}$ method), then by definition of g they are mapped to $\mathsf{s}_1 = (\mathsf{e}, \mathtt{aNLR}_n)$ and $\mathsf{s}_2 = (\mathsf{e}, \mathtt{aNRW}_n)$ using the same `Put` event and so $\mathsf{v}'_{\mathsf{R}}(\mathsf{s}'_1) = \mathsf{v}_{\mathsf{R}}(\mathsf{s}_1) = \mathsf{v}_{\mathsf{W}}(\mathsf{s}_2) = \mathsf{v}'_{\mathsf{W}}(\mathsf{s}'_2)$ since v_{R} and v_{W} are well-formed.

We define $\mathsf{rf}' \triangleq \bigcup_n \mathsf{rf}'^n$ and $\mathsf{mo}' \triangleq \bigcup_{x,n} \mathsf{mo}'^n_x$ from rf and mo as follows:

$$\mathsf{rf}'^n \triangleq \{(w, r) \mid r \in \mathcal{G}'.\mathcal{R}^n \wedge (g(w), g(r)) \in \mathsf{rf}\}$$

$$\mathsf{mo}'^n_x \triangleq \left\{(w_1, w_2) \;\middle|\; (g(w_1), g(w_2)) \in \mathsf{mo}_{x_n}\right\}$$

As an illustration, Fig. 26 shows how v'_{R}, v'_{W}, rf', and mo' are defined on a small program. The relations naturally carry over from the implementation to the client program. In general, we show that rf' and mo' are indeed well formed (see 3.4).

It is straightforward to check that $\mathsf{rf}'^n \subseteq \mathcal{G}'.\mathcal{W}^n \times \mathcal{G}'.\mathcal{R}^n$. If $(\mathsf{s}'_1, \mathsf{s}'_2) \in \mathsf{rf}'^n$, then $(g(\mathsf{s}'_1), g(\mathsf{s}'_2)) \in \mathsf{rf}$ and $\mathsf{v}'_{\mathsf{W}}(\mathsf{s}'_1) = \mathsf{v}_{\mathsf{W}}(g(\mathsf{s}'_1)) = \mathsf{v}_{\mathsf{R}}(g(\mathsf{s}'_2)) = \mathsf{v}'_{\mathsf{R}}(\mathsf{s}'_2)$.

We argue that if $\mathsf{s}'_2 \notin \mathtt{img}(\mathsf{rf}'^n)$ then $g(\mathsf{s}'_2) \notin \mathtt{img}(\mathsf{rf})$. This might not be obvious since rf is bigger, as it has for instance statements about the $\bot_n$ locations. The reason is that, for each node n and sv location x, the relation g^{-1} is total and functional on $\mathcal{G}.\mathcal{W}_{x_n}$, i.e. every write subevent in the implementation (outside those on the dummy locations $\bot_n$) is associated with a write subevent of the sv library. This can be checked by considering I_{SV} and the different cases of the definition of g. Thus if $(\mathsf{s}_1, g(\mathsf{s}'_2)) \in \mathsf{rf}$ there is s'_1 such that $g(\mathsf{s}'_1) = \mathsf{s}_1$ and $\mathsf{s}'_2 \in \mathtt{img}(\mathsf{rf}'^n)$. So for a subevent $\mathsf{s}'_2 \notin \mathtt{img}(\mathsf{rf}'^n)$ we have $g(\mathsf{s}'_2) \notin \mathtt{img}(\mathsf{rf})$ and $\mathsf{v}'_{\mathsf{R}}(\mathsf{s}'_2) = \mathsf{v}_{\mathsf{R}}(g(\mathsf{s}'_2)) = 0$.

We also need to check that each mo'^n_x is a strict total order on $\mathcal{G}'.\mathcal{W}^n_x$. This is simply because for all $\mathsf{s}' \in \mathcal{G}'.\mathcal{W}^n_x$ we have $g(\mathsf{s}') \in \mathcal{G}.\mathcal{W}_{x_n}$, and we know mo is a strict total order on $\mathcal{G}.\mathcal{W}_{x_n}$.

Now that v'_{R}, v'_{W}, rf', and mo' are defined, the derived relations pf', rb', iso', $\mathsf{rf}'_{\mathsf{e}}$, and $\mathsf{so}' \triangleq \mathsf{iso}' \cup \mathsf{rf}'_{\mathsf{e}} \cup \mathsf{pf}' \cup \mathsf{rb}' \cup \mathsf{mo}'$ are also available (see 3.4). We then need to prove that $g(\mathsf{so}') \subseteq \mathsf{hb}$, which can be checked component by component.

- If $(\mathsf{s}'_1, \mathsf{s}'_2) \in \mathsf{iso}'$, then there is n and $\mathsf{e}' = (t, \_, (\mathtt{Bcast}_{\mathsf{sv}}, (x, \_, \{\ldots; n; \ldots\}), \_)) \in E'$ such that $\mathsf{s}'_1 = (\mathsf{e}', \mathtt{aNLR}_n)$ and $\mathsf{s}'_2 = (\mathsf{e}', \mathtt{aNRW}_n)$. By definition of g, there is $\mathsf{e} = (t, \_, (\mathtt{Put}, (x_n, x_{\mathsf{n}(t)}, \_), \_)) \in f^{-1}(\mathsf{e}')$ such that $g(\mathsf{s}'_1) = (\mathsf{e}, \mathtt{aNLR}_n)$ and $g(\mathsf{s}'_2) = (\mathsf{e}, \mathtt{aNRW}_n)$. And by definition of $\mathcal{G}.\mathsf{iso}$, we have $(g(\mathsf{s}'_1), g(\mathsf{s}'_2)) \in \mathcal{G}.\mathsf{iso} \subseteq \mathsf{so} \subseteq \mathsf{hb}$.
- By definition $g(\mathsf{rf}') \subseteq \mathsf{rf}$. We want to show $g(\mathsf{rf}'_{\mathsf{e}}) \subseteq \mathsf{rf}_{\mathsf{e}} \subseteq \mathsf{so} \subseteq \mathsf{hb}$. Note that for all node n and subevent $\mathsf{s}' \in \mathcal{G}'.\mathcal{R}^n \cup \mathcal{G}'.\mathcal{W}^n$, the function g maps to a subevent using the same stamp: $\mathsf{s}'.a = g(\mathsf{s}').a$. Also, from the abstraction f, we know that g preserves the program order:

if $(s'_1, s'_2) \in \mathrm{po}'$, then $(g(s'_1), g(s'_2)) \in \mathrm{po}$. Thus g preserves the internal/external distinction: $g(\mathrm{rf}'_i) \subseteq \mathrm{rf}_i$ and $g(\mathrm{rf}'_e) \subseteq \mathrm{rf}_e$, which implies $g(\mathrm{rf}'_e) \subseteq \mathrm{hb}$.

- If $(s'_1, s'_2) \in \mathrm{pf}'$, then by definition there is d, n, $e'_1 = (\_, \_, (\mathtt{Bcast}_{\mathrm{sv}}, (\_, d, \{\ldots; n; \ldots\}), \_))$, and $e'_2 = (\_, \_, (\mathtt{Wait}_{\mathrm{sv}}, (d), \_))$ such that $(e'_1, e'_2) \in \mathrm{po}'$, $s'_1 = (e'_1, \mathtt{aNLR}_n)$, and $s'_2 = (e'_2, \mathtt{aWT})$. From the abstraction f and the definition of g, there is $e_1 = (\_, \_, (\mathtt{Put}, (\_, \_, d), \_))$ and $e_2 = (\_, \_, (\mathtt{Wait}, (d), \_))$ such that $(e_1, e_2) \in \mathrm{po}$, $g(s'_1) = (e_1, \mathtt{aNLR}_n)$, and $g(s'_2) = (e_2, \mathtt{aWT})$. This case is illustrated in Fig. 27. We have $(e_1, \mathtt{aNLR}_n) \xrightarrow{\mathcal{G}.\mathrm{iso}} (e_1, \mathtt{aNRW}_n) \xrightarrow{\mathcal{G}.\mathrm{pfp}} (e_2, \mathtt{aWT})$, and so $((e_1, \mathtt{aNLR}_n), (e_1, \mathtt{aNLR}_n)) \in \mathcal{G}.\mathrm{ib}$. Since $(e_1, \mathtt{aNLR}_n) \in \mathcal{G}.\mathtt{aNLR}_n \subseteq \mathcal{G}.\mathtt{Inst}$, we have $(g(s'_1), g(s'_2)) \in ([\mathcal{G}.\mathtt{Inst}]; \mathcal{G}.\mathrm{ib}) \subseteq \mathrm{so} \subseteq \mathrm{hb}$.
- We can check that $g(\mathrm{rb}') \subseteq \mathrm{rb}$, which intuitively comes from the correspondence between $\mathrm{rf}'/\mathrm{mo}'$ and rf/mo. If $(s'_1, s'_2) \in \mathrm{rb}^n$, then by definition there is x such that $s'_1 \in \mathcal{G}'.\mathcal{R}^n$, $s'_2 \in \mathcal{G}'.\mathcal{W}^n_x$, $\mathtt{loc}(s'_1) = x = \mathtt{loc}(s'_2)$, and either $(s'_1, s'_2) \in ((\mathrm{rf}'^n)^{-1}; \mathrm{mo}'^n_x)$ or $s'_2 \notin \mathtt{img}(\mathrm{rf}'^n)$. Since $\mathcal{G}'.\mathcal{R}^n \cap \mathcal{G}'.\mathcal{W}^n = \emptyset$, as the library does not have any read-modify-write method, we also know $s'_1 \neq s'_2$ and by definition of g that $g(s'_1) \neq g(s'_2)$.
  - If there is s'_3 such that $(s'_3, s'_1) \in \mathrm{rf}'^n$ and $(s'_3, s'_2) \in \mathrm{mo}'^n_x$, then by definition $(g(s'_3), g(s'_1)) \in \mathrm{rf}$ and $(g(s'_3), g(s'_2)) \in \mathrm{mo}_{x_n}$, so $(g(s'_1), g(s'_2)) \in \mathrm{rb}$.
  - If $s'_2 \notin \mathtt{img}(\mathrm{rf}'^n)$ then $g(s'_2) \notin \mathtt{img}(\mathrm{rf})$ (proved earlier) and $(g(s'_1), g(s'_2)) \in \mathrm{rb}$.

  And so $g(\mathrm{rb}') \subseteq \mathrm{rb} \subseteq \mathrm{so} \subseteq \mathrm{hb}$.
- Finally we have $g(\mathrm{mo}') \subseteq \mathrm{mo} \subseteq \mathrm{so} \subseteq \mathrm{hb}$ by definition.

Thus $g(\mathrm{so}') \subseteq \mathrm{hb}$.

Lastly, we are left to prove that $[\mathtt{aCR}]; (\mathrm{po}'^{-1} \cap \mathrm{rb}'); [\mathtt{aCW}] = \emptyset$. This comes from the fact that $[\mathtt{aCR}]; (\mathrm{po}^{-1} \cap \mathrm{rb}); [\mathtt{aCW}] \subseteq \mathrm{rb}_i \subseteq \mathrm{ib}$, $[\mathtt{aCW}]; \mathrm{po}; [\mathtt{aCR}] \subseteq \mathrm{ippo} \subseteq \mathrm{ib}$, and $g(\mathrm{rb}') \subseteq \mathrm{rb}$ (proved earlier). So if $(s'_1, s'_2) \in [\mathtt{aCR}]; (\mathrm{po}'^{-1} \cap \mathrm{rb}'); [\mathtt{aCW}]$, we have $(g(s'_1), g(s'_2)) \in [\mathtt{aCR}]; (\mathrm{po}^{-1} \cap \mathrm{rb}); [\mathtt{aCW}] \subseteq \mathrm{ib} \cap \mathrm{ib}^{-1} = \emptyset$ which is not possible, since we know ib is transitive and irreflexive.

Thus $\mathcal{G}'$ is sv-consistent and the implementation I_{SV} is locally sound. □

Corollary G.2. *The implementation I_{SV} is sound.*

G.2 msw Library

Theorem G.3. *Given a function* `size`*, the implementation $I^{\mathtt{size}}_{\mathrm{MSW}}$ (§G.6.1) of the msw library into rdma$^{\mathrm{WAIT}}$ given in the paper is locally sound.*

Proof. We assume an $\{\mathrm{RDMA}^{\mathrm{WAIT}}\}$-consistent execution $\mathcal{G} = \langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle$ which is abstracted via f to $\langle E', \mathrm{po}'\rangle$ that uses the msw library, i.e. $\mathsf{abs}^f_{I^{\mathtt{size}}_{\mathrm{MSW}}, \mathrm{MSW}}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$ holds. We need to provide $\mathtt{stmp}'$, so', and $g : \langle E', \mathrm{po}', \mathtt{stmp}'\rangle.\mathrm{SEvent} \to \mathcal{G}.\mathrm{SEvent}$ respecting some conditions. From $\langle E', \mathrm{po}'\rangle$, we simply take $\mathtt{stmp}' = \mathtt{stmp}_{\mathrm{MSW}}$.

Since the implementation $I^{\mathtt{size}}_{\mathrm{MSW}}$ maps events that do not respect the `size` function to non-terminating loops, the abstraction f tells us that every event in E' does respect the `size`.

Since $\mathcal{G}$ is $\{\mathrm{RDMA}^{\mathrm{WAIT}}\}$-consistent, it means $(\mathrm{ppo} \cup \mathrm{so})^+ \subseteq \mathrm{hb}$, hb is transitive and irreflexive, and $\mathcal{G}$ is $\mathrm{RDMA}^{\mathrm{WAIT}}$-consistent. Firstly, it means that $\langle E, \mathrm{po}\rangle$ respects nodes. From the properties of $\mathsf{abs}^f_{I^{\mathtt{size}}_{\mathrm{MSW}}, \mathrm{MSW}}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$, we can easily see that it implies $\langle E, \mathrm{po}\rangle$ respects nodes, as the implementation locations are mapped to the same nodes: $\mathtt{n}(x_1) = \ldots = \mathtt{n}(x_{\mathtt{size}(x)}) = \mathtt{n}(x)$. Secondly, there exists well-formed v_R, v_W, rf, mo, and nfo such that ib is irreflexive, $\mathtt{stmp} = \mathtt{stmp}_{\mathrm{RL}}$, and $\mathrm{so} = \mathrm{iso} \cup \mathrm{rf}_e \cup \mathrm{pfg} \cup \mathrm{nfo} \cup \mathrm{rb} \cup \mathrm{mo} \cup ([\mathtt{Inst}]; \mathrm{ib})$.

We define g as follows.

- For an event $e' = (t, \_, (\mathtt{Write}^{\mathrm{MSW}}, (x, \widetilde{v}), ()))$, we choose $g(e', \mathtt{aCW}) = (e, \mathtt{aCW})$ with $e = (t, \_, (\mathtt{Write}, (x_0, \mathtt{hash}(\widetilde{v})), ())) \in f^{-1}(e')$.

- For an event $\mathtt{e}' = (t, \_, (\mathtt{TryRead}^{\mathrm{MSW}}, (x), \widetilde{v}))$, we choose $g(\mathtt{e}', \mathtt{aCR}) = (\mathtt{e}, \mathtt{aCR})$ with $\mathtt{e} = (t, \_, (\mathtt{Read}, (x_0), v_0))$ and $v_0 = \mathtt{hash}(\widetilde{v})$.
- For an event $\mathtt{e}' = (t, \_, (\mathtt{TryRead}^{\mathrm{MSW}}, (x), \bot))$, we choose $g(\mathtt{e}', \mathtt{aWT}) = (\mathtt{e}, \mathtt{aCR})$ with $\mathtt{e} = (t, \_, (\mathtt{Read}, (x_0), v_0)))$.
- For an event $\mathtt{e}' = (t, \_, (\mathtt{Put}^{\mathrm{MSW}}, (x, y, d), ()))$, we choose $g(\mathtt{e}', \mathtt{aNLR}_{\mathtt{n}(x)}) = (\mathtt{e}, \mathtt{aNLR}_{\mathtt{n}(x_0)})$ and $g(\mathtt{e}', \mathtt{aNRW}_{\mathtt{n}(x)}) = (\mathtt{e}, \mathtt{aNRW}_{\mathtt{n}(x_0)})$ with $\mathtt{e} = (t, \_, (\mathtt{Put}, (x_0, y_0, d), ()))$.
- For an event $\mathtt{e}' = (t, \_, (\mathtt{Get}^{\mathrm{MSW}}, (x, y, d), ()))$, we choose $g(\mathtt{e}', \mathtt{aNRR}_{\mathtt{n}(y)}) = (\mathtt{e}, \mathtt{aNRR}_{\mathtt{n}(y_0)})$ and $g(\mathtt{e}', \mathtt{aNLW}_{\mathtt{n}(y)}) = (\mathtt{e}, \mathtt{aNLW}_{\mathtt{n}(y_0)})$ with $\mathtt{e} = (t, \_, (\mathtt{Get}, (x_0, y_0, d), ()))$.
- For an event $\mathtt{e}' = (t, \_, (\mathtt{Wait}^{\mathrm{MSW}}, (d), ()))$, we choose $g(\mathtt{e}', \mathtt{aWT}) = (\mathtt{e}, \mathtt{aWT})$ with $\mathtt{e} = (t, \_, (\mathtt{Wait}, (d), ()))$.

This definition of g clearly preserves to (first property to check) using the identity function, since aCR and aWT have the same relation to other stamps.

Note that for every location x, every write subevent on x_0 in the implementation is in the image of g.

Now we need to find so′ such that $g(\mathtt{so}') \subseteq \mathtt{hb}$ and such that $\mathcal{G}' = \langle E', \mathrm{po}', \mathtt{stmp}', \mathtt{so}', \_\rangle$ is MSW-consistent. For this, we need to choose well-formed $\mathtt{v}'_{\mathsf{R}}$, $\mathtt{v}'_{\mathsf{W}}$, rf′, mo′, and nfo′. We define $\mathtt{v}'_{\mathsf{R}}(\mathtt{s}') = \mathtt{hash}^{-1}(\mathtt{v}_{\mathsf{R}}(g(\mathtt{s}')))$, and similarly for $\mathtt{v}'_{\mathsf{W}}$. E.g., when the implementation of $\mathtt{Put}^{\mathrm{MSW}}(x, y, d)$ reads (and writes) the values $\mathtt{hash}((v_1, \ldots, v_{\mathtt{size}(x)})), v'_1, \ldots, v'_{\mathtt{size}(x)}$, we pretend $\mathtt{Put}^{\mathrm{MSW}}(x, y, d)$ actually reads $(v_1, \ldots, v_{\mathtt{size}(x)})$, even if the following data is corrupted and does not correspond to the hash. For the sake of simplicity, we assume that $\mathtt{hash}(\widetilde{0}) = 0$, or equivalently that the hash locations can be initialised to $\mathtt{hash}(\widetilde{0})$. For $\mathtt{Write}^{\mathrm{MSW}}$ events, the $\mathtt{v}'_{\mathsf{W}}$ function matches the values written, as required. For a succeeding $\mathtt{TryRead}^{\mathrm{MSW}}$ event, the if-then-else construct ensures that the value returned is the inverse of the hash, matching the $\mathtt{v}'_{\mathsf{R}}$ function as required.

We then define $\mathtt{rf}' = \{(\mathtt{s}'_1, \mathtt{s}'_2) \mid (g(\mathtt{s}'_1), g(\mathtt{s}'_2)) \in \mathtt{rf}\}$, $\mathtt{mo}' = \{(\mathtt{s}'_1, \mathtt{s}'_2) \mid (g(\mathtt{s}'_1), g(\mathtt{s}'_2)) \in \mathtt{mo}\}$, and $\mathtt{nfo}' = \{(\mathtt{s}'_1, \mathtt{s}'_2) \mid (g(\mathtt{s}'_1), g(\mathtt{s}'_2)) \in \mathtt{nfo}\}$, and they are well-formed:

- If $(\mathtt{s}'_1, \mathtt{s}'_2) \in \mathtt{rf}'$, then we have $\mathtt{v}'_{\mathsf{W}}(\mathtt{s}'_1) = \mathtt{hash}^{-1}(\mathtt{v}_{\mathsf{W}}(g(\mathtt{s}'_1))) = \mathtt{hash}^{-1}(\mathtt{v}_{\mathsf{R}}(g(\mathtt{s}'_2))) = \mathtt{v}'_{\mathsf{R}}(\mathtt{s}'_2)$. If $\mathtt{s}'_2 \notin \mathtt{img}(\mathtt{rf}')$ on location x, then since every write subevent on x_0 in the implementation is in the image of g we have $g(\mathtt{s}'_2) \notin \mathtt{img}(\mathtt{rf})$ and $\mathtt{v}'_{\mathsf{R}}(\mathtt{s}'_2) = \mathtt{hash}^{-1}(\mathtt{v}_{\mathsf{R}}(g(\mathtt{s}'_2))) = \mathtt{hash}^{-1}(0) = \widetilde{0}$.
- $\mathtt{mo}'_x$ is total on $\mathcal{G}'.\mathcal{W}_x$ since $\mathtt{mo}_{x_0}$ is total on $\mathcal{G}.\mathcal{W}_{x_0}$ and every write on x_0 is in the image of g.
- If $\mathtt{t}(\mathtt{s}'_1) = \mathtt{t}(\mathtt{s}'_2)$ and $(\mathtt{s}'_1, \mathtt{s}'_2) \in \mathcal{G}'.\mathtt{aNLR}_n \times \mathcal{G}'.\mathtt{aNLW}_n$ (resp. $\mathcal{G}'.\mathtt{aNRR}_n \times \mathcal{G}'.\mathtt{aNRW}_n$) then $\mathtt{t}(g(\mathtt{s}'_1)) = \mathtt{t}(\mathtt{s}'_1) = \mathtt{t}(\mathtt{s}'_2) = \mathtt{t}(g(\mathtt{s}'_2))$ and $(g(\mathtt{s}'_1), g(\mathtt{s}'_2)) \in \mathcal{G}.\mathtt{aNLR}_n \times \mathcal{G}.\mathtt{aNLW}_n$. So $(g(\mathtt{s}'_1), g(\mathtt{s}'_2)) \in \mathtt{nfo} \cup \mathtt{nfo}^{-1}$ and we also have $(\mathtt{s}'_1, \mathtt{s}'_2) \in \mathtt{nfo}' \cup \mathtt{nfo}'^{-1}$.

It is straightforward to see that $g(\mathtt{rf}') \subseteq \mathtt{rf}$, $g(\mathtt{mo}') \subseteq \mathtt{mo}$, $g(\mathtt{nfo}') \subseteq \mathtt{nfo}$, $g(\mathtt{pfg}') \subseteq \mathtt{pfg}$, $g(\mathtt{pfp}') \subseteq \mathtt{pfp}$, $g(\mathtt{ippo}') \subseteq \mathtt{ippo}$, $g(\mathtt{ppo}') \subseteq \mathtt{ppo}$, $g(\mathtt{rf}'_{\mathsf{e}}) \subseteq \mathtt{rf}_{\mathsf{e}}$, and $g(\mathtt{iso}') \subseteq \mathtt{iso}$. The only non-obvious relation might be $g(\mathtt{rb}') \subseteq \mathtt{rb}$. Let $(\mathtt{s}'_1, \mathtt{s}'_2) \in \mathtt{rb}'$:

- If $\mathtt{s}'_2 \notin \mathtt{img}(\mathtt{rf}')$, as mentioned earlier we have $g(\mathtt{s}'_2) \notin \mathtt{img}(\mathtt{rf})$ and thus $(g(\mathtt{s}'_1), g(\mathtt{s}'_2)) \in \mathtt{rb}$.
- If there is $\mathtt{s}'_3$ such that $(\mathtt{s}'_3, \mathtt{s}'_1) \in \mathtt{rf}'$ and $(\mathtt{s}'_3, \mathtt{s}'_2) \in \mathtt{mo}'$, then we have $(g(\mathtt{s}'_3), g(\mathtt{s}'_1)) \in \mathtt{rf}$ and $(g(\mathtt{s}'_3), g(\mathtt{s}'_2)) \in \mathtt{mo}$, and so $(g(\mathtt{s}'_1), g(\mathtt{s}'_2)) \in \mathtt{rb}$.

And of course $g(\mathtt{rb}'_{\mathsf{i}}) \subseteq \mathtt{rb}_{\mathsf{i}}$ also holds since the stamps are preserved.

Thus we have $g(\mathtt{ib}') \subseteq \mathtt{ib}$, implying ib′ is irreflexive, $g(\mathtt{so}') \subseteq \mathtt{so} \subseteq \mathtt{hb}$, and we have $\mathcal{G}' = \langle E', \mathrm{po}', \mathtt{stmp}', \mathtt{so}', \_\rangle$ is MSW-consistent. □

Corollary G.4. *The implementation $I^{\mathtt{size}}_{\mathrm{MSW}}$ is sound.*

G.3 BAL Library

THEOREM G.5. *Given a function* b*, the implementation* $I^{\mathsf{b}}_{\mathsf{BAL}}$ *of the* BAL *library into* SV *given in the paper is locally sound.*

PROOF. We assume an $\{\mathrm{sv}\}$-consistent execution $\mathcal{G} = \langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle$ which is abstracted via f to $\langle E', \mathrm{po}'\rangle$ that uses the BAL library, i.e. $\mathsf{abs}^{f}_{I^{\mathsf{b}}_{\mathsf{BAL}},\mathrm{BAL}}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$ holds. We need to provide $\mathtt{stmp}'$, so', and $g : \langle E', \mathrm{po}', \mathtt{stmp}'\rangle.\mathrm{SEvent} \to \mathcal{G}.\mathrm{SEvent}$ respecting some conditions. From $\langle E', \mathrm{po}'\rangle$, we simply take $\mathtt{stmp}' = \mathtt{stmp}_{\mathsf{BAL}}$.

Since $\mathcal{G}$ is $\{\mathrm{sv}\}$-consistent, it means $(\mathrm{ppo} \cup \mathrm{so})^+ \subseteq \mathrm{hb}$, hb is transitive and irreflexive, and $\mathcal{G}$ is sv-consistent. Firstly, it means that for all thread t we have $\mathrm{po}|_t$ is a strict total order. From the properties of $\mathsf{abs}^{f}_{I^{\mathsf{b}}_{\mathsf{BAL}},\mathrm{BAL}}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$, we can easily see that it implies $\mathrm{po}'|_t$ is also a strict total order. Secondly, $\mathtt{stmp} = \mathtt{stmp}_{\mathsf{SV}}$ and there exists well-formed v_{R}, v_{W}, rf, and mo such that $[\mathtt{aCR}]; (\mathrm{po}^{-1} \cap \mathrm{rb}); [\mathtt{aCW}] = \emptyset$ and $\mathrm{so} = \mathrm{iso} \cup \mathrm{rf}_{\mathsf{e}} \cup \mathrm{pf} \cup \mathrm{rb} \cup \mathrm{mo}$.

By definition of the abstraction, for each event $\mathsf{e}' = (t, \_, (\mathsf{BAR}_{\mathrm{BAL}}, (x), ()))$ $\in E'$ we have $\langle\langle(), 0\rangle, \langle E, \mathrm{po}\rangle|_{f^{-1}(\mathsf{e}')}\rangle \in [\![I^{\mathsf{b}}_{\mathsf{BAL}}(t, \mathsf{BAR}_{\mathrm{BAL}}, (x))]\!]_t$. Since by definition $[\![\mathtt{loop}\{()\}]\!]_t = \emptyset$, we necessarily have $t \in \mathsf{b}(x)$. We note $s_n = \{\mathsf{n}(t_i) \mid t_i \in \mathsf{b}(x)\}$ the nodes involved in the barrier. The size of $E|_{f^{-1}(\mathsf{e}')}$ depends on how many times the loops read the locations of other threads, but this subgraph contains at least the global fence $\mathsf{e}_{GF} = (t, \_, (\mathsf{GF}_{\mathrm{sv}}, (s_n), ()))$, the first read $\mathsf{e}_{FR} = (t, \_, (\mathsf{Read}_{\mathrm{sv}}, (x_t), (v)))$, the write $\mathsf{e}_W = (t, \_, (\mathtt{Write}_{\mathrm{sv}}, (x_t, v+1), ()))$, and the last read $\mathsf{e}_{LR} = (t, \_, (\mathsf{Read}_{\mathrm{sv}}, (x_{t_k}), (v')))$ with $v' > v$, such that for any other event $\mathsf{e}_0 \in E|_{f^{-1}(\mathsf{e}')}$ besides these four, we have $\mathsf{e}_{GF} \xrightarrow{\mathrm{po}} \mathsf{e}_{FR} \xrightarrow{\mathrm{po}} \mathsf{e}_W \xrightarrow{\mathrm{po}} \mathsf{e}_0 \xrightarrow{\mathrm{po}} \mathsf{e}_{LR}$. If all threads are not on the same nodes, we also have a broadcast event $\mathsf{e}_{BR} = (t, \_, (\mathtt{Bcast}_{\mathrm{sv}}, (x_t, \_, (s_n \setminus \{\mathsf{n}(t)\})), ()))$ with $\mathsf{e}_W \xrightarrow{\mathrm{po}} \mathsf{e}_{BR}$.

We define g as expected: $g(\mathsf{e}', \mathtt{aGF}_n) \triangleq (\mathsf{e}_{GF}, \mathtt{aGF}_n)$ for $n \in s_n$, and $g(\mathsf{e}', \mathtt{aCR}) \triangleq (\mathsf{e}_{LR}, \mathtt{aCR})$. This clearly preserves to (first property of g) using the identity function.

We also define $o(\mathsf{e}') \triangleq v + 1$, i.e. the value written by e_W. For a location x, we note $c_x \triangleq \mathtt{max}_{(\mathsf{e}' \in E'_x)} o(\mathsf{e}')$ the maximum value attributed to a barrier call on x. We are forced to take the only valid synchronisation order $\mathrm{so}' = \bigcup_{x \in \mathrm{Loc}} \bigcup_{1 \le i \le c_x} \left\{((\mathsf{e}'_1, \mathtt{aGF}_n), (\mathsf{e}'_2, \mathtt{aCR})) \;\middle|\; \mathsf{e}'_1, \mathsf{e}'_2 \in (E'_x \cap o^{-1}(i))\right\}$ and we need to show that $g(\mathrm{so}') \subseteq \mathrm{hb}$ and that $\mathcal{G}' = \langle E', \mathrm{po}', \mathtt{stmp}', \mathrm{so}', \_\rangle$ is BAL-consistent.

Let us start with the conditions on $\mathcal{G}'$, where we need to check that c_x and o respect some properties. By definition, for $\mathsf{e}' \in E'_x$ we have $1 \le o(\mathsf{e}) \le c_x$. For a thread $t \notin \mathsf{b}(x)$, we have seen that the implementation prevents any event on x. For a thread $t \in \mathsf{b}(x)$, we will show $\#(E'_x|_t) = c_x$ by checking that every number from 1 to c_x is attributed once.

Note that, for a given thread t and location x, since E' only contains barrier calls, only the events e_W of the form $(t, \_, (\mathtt{Write}_{\mathrm{sv}}, (x_t, v+1), ()))$ are able to modify the value of x_t on node $\mathsf{n}(t)$[5]. Similarly, the value of x_t on another node can only be modified by a broadcast event from the thread t, thus copying the value written by an e_W event.

Firstly, let us show c_x is attributed on every participating thread t. By definition of c_x there is e'_0 on thread t_0 writing c_x. From the definition of the implementation of e'_0, there is a loop that only finishes when reading x_t with value $v' \ge c_x$. This value v' can only be is created by an event $\mathsf{e}' \in E'_x|_t$, and we have $o(\mathsf{e}') = v' \ge x_k$. Since c_x is defined as the maximum of such values, we have $v' = c_x$ and c_x is attributed on t.

Secondly, let us show that if $v+2$ is attributed, then $v+1$ is attributed. This is simply because if $o(\mathsf{e}'_2) = v+2$, i.e. the implementation of e'_2 writes $v+2$, then the initial read events $\mathsf{e}_{FR} = (t, \_, (\mathsf{Read}_{\mathrm{sv}}, (x_t), (v+1))) \in f^{-1}(\mathsf{e}'_2)$ reads the value $v+1$. As before, this value can only be is created by an event $\mathsf{e}'_1 \in E'_x|_t$, and we have $o(\mathsf{e}'_1) = v+1$.

[5]This is why the broadcast event must *not* overwrite x_t with itself on node $\mathsf{n}(t)$.

Thirdly, let us show that if $e'_1, e'_2 \in E'_x$ and $(e'_1, e'_2) \in \mathrm{po}'$ then $o(e'_1) < o(e'_2)$. By contradiction, let us assume $(e'_1, e'_2) \in \mathrm{po}'|_{\mathrm{imm}}$ the first pair (in $\mathrm{po}'|_{E'_x}$ order) such that $o(e'_1) = i+1 \geq j+1 = o(e'_2)$. As previously, their implementations have events $e^1_{FR} = (t, \_, (\mathtt{Read}_{\mathrm{sv}}, (x_t), (i))) \in f^{-1}(e'_1)$, $e^1_W = (t, \_, (\mathtt{Write}_{\mathrm{sv}}, (x_t, i{+}1), ())) \in f^{-1}(e'_1)$, and $e^2_{FR} = (t, \_, (\mathtt{Read}_{\mathrm{sv}}, (x_t), (j))) \in f^{-1}(e'_2)$, with $e^1_{FR} \xrightarrow{\mathrm{po}} e^1_W \xrightarrow{\mathrm{po}} e^2_{FR}$. Let $s_2 = (e^2_{FR}, \mathtt{aCR})$ and $s_1 = (e^1_W, \mathtt{aCW})$. Since we know $[\mathtt{aCR}]; (\mathrm{po}^{-1} \cap \mathrm{rb}); [\mathtt{aCW}] = \emptyset$, showing $(s_2, s_1) \in \mathrm{rb}$ would be a contradiction.

- If $(\_, s_2) \notin \mathrm{rf}$ (i.e. $j = 0$), then $(s_2, s_1) \in \mathrm{rb}$ is a contradiction.
- If $(s_3, s_2) \in \mathrm{rf}$ with $(s_2, s_3) \in \mathrm{po}$, then since s_3 uses the stamp $\mathtt{aCW}$ we have $(s_2, s_3) \in \mathrm{ppo} \subseteq \mathrm{hb}$ and $(s_3, s_2) \in \mathrm{rf_e} \subseteq \mathrm{so} \subseteq \mathrm{hb}$. Thus we have an hb cycle, which is a contradiction.
- If $(s_3, s_2) \in \mathrm{rf}$ with $(s_3, s_2) \in \mathrm{po}$, since $(e'_1, e'_2) \in \mathrm{po}'|_{\mathrm{imm}}$ there is no write in-between s_1 and s_2 and thus $(s_3, s_1) \in \mathrm{po}$. Since $\mathrm{mo}^{\mathrm{n}(t)}_{x_t}$ is included in hb and only uses the stamp $\mathtt{aCW}$, it coincides with po and so $(s_3, s_1) \in \mathrm{mo}$ and $(s_2, s_1) \in \mathrm{rb}$ is a contradiction.

Thus $(e'_1, e'_2) \in \mathrm{po}'$ implies $o(e'_1) < o(e'_2)$.

By combining the pieces above, every number from 1 to c_x is attributed exactly once and $\#(E_x|_t) = c_x$. This concludes the properties on $\mathcal{G}'$ and we have that $\mathcal{G}' = \langle E', \mathrm{po}', \mathtt{stmp}', \mathrm{so}', \_\rangle$ is BAL-consistent.

Finally, the last part of the proof is to check that $g(\mathrm{so}') \subseteq \mathrm{hb}$. Let us assume $s'_1 = (e'_1, \mathtt{aGF}_n)$, $s'_2 = (e'_2, \mathtt{aCR})$, and $(s'_1, s'_2) \in \mathrm{so}'$ for some x and i on threads t_1 and t_2. So $e'_1, e'_2 \in E'_x$ and $o(e'_1) = o(e'_2) = i$. By definition of the implementation and g we have $e^1_{GF} \xrightarrow{\mathrm{po}} e^1_W$ in $f^{-1}(e'_1)$ with $g(s'_1) = (e^1_{GF}, \mathtt{aGF}_n)$, as well as $e^2 \xrightarrow{\mathrm{po}} e^2_{LR}$ in $f^{-1}(e'_2)$ with $g(s'_2) = (e_{LR}, \mathtt{aCR})$ and $e^2 = (t_2, \_, (\mathtt{Read}_{\mathrm{sv}}, (x_{t_1}), (v')))$ is the last read of the loop for thread t_1 reading a value $v' \geq i$. In the very specific case where $t_1 = t_2$, we have $(g(s'_1), g(s'_2)) \in \mathrm{ppo} \subseteq \mathrm{hb}$. Otherwise, let $s_1 = (e^1_W, \mathtt{aCW})$ and $s_2 = (e^2, \mathtt{aCR})$. By definition of to we have $(g(s'_1), s_1) \in \mathrm{ppo} \subseteq \mathrm{hb}$ and $(s_2, g(s'_2)) \in \mathrm{ppo} \subseteq \mathrm{hb}$, so we are left to prove $(s_1, s_2) \in \mathrm{hb}$.

If t_1 and t_2 are on the same node, there is no broadcast involved. If s_2 reads from s_1 (i.e. $v' = i$), then we immediately have $(s_1, s_2) \in \mathrm{rf_e} \subseteq \mathrm{so} \subseteq \mathrm{hb}$. Else (i.e. $v' > i$) s_2 reads from some subevent $(e^3, \mathtt{aCW})$ on thread t_1 in the implementation of a later barrier, with $o(f(e^3)) = v'$. From the properties of o and the abstraction we have $(e^1_W, e^3) \in \mathrm{po}$. So $(s_1, s_2) \in (\mathrm{ppo}; \mathrm{rf_e}) \subseteq \mathrm{hb}$.

If t_1 and t_2 are on different nodes n_1 and n_2, the reasoning is similar except a broadcast from t_1 bridges the gap. There is $e_B = (t_1, \_, (\mathtt{Bcast}_{\mathrm{sv}}, (x_{t_1}, \_, \{\ldots; n_2; \ldots\}), ()))$ such that $((e_B, \mathtt{aNLR}_{n_2}), (e_B, \mathtt{aNRW}_{n_2})) \in \mathrm{iso} \subseteq \mathrm{hb}$, $((e_B, \mathtt{aNRW}_{n_2}), s_2) \in \mathrm{rf_e} \subseteq \mathrm{hb}$, and $(e_B, \mathtt{aNRW}_{n_2})$ reads the same value v'. We fall back to the previous case: if $(e_B, \mathtt{aNRW}_{n_2})$ reads from s_1 we have $(s_1, (e_B, \mathtt{aNRW}_{n_2})) \in \mathrm{rf_e} \subseteq \mathrm{hb}$; if it reads from a later write we have $(s_1, (e_B, \mathtt{aNRW}_{n_2})) \in (\mathrm{ppo}; \mathrm{rf_e}) \subseteq \mathrm{hb}$. In all cases, we have $(s_1, s_2) \in \mathrm{hb}$. □

Corollary G.6. *The implementation $I^{\mathsf{b}}_{\mathsf{BAL}}$ is sound.*

G.4 RBL Library

Theorem G.7. *Given the functions* wthd *and* rthd *and a size S, the implementation $I^{\mathtt{wthd},\mathtt{rthd}}_{\mathrm{S,RBL}}$ of the RBL library into SV given in the paper is locally sound.*

Proof. We assume an {sv}-consistent execution $\mathcal{G} = \langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle$ which is abstracted via f to $\langle E', \mathrm{po}'\rangle$ that uses the RBL library, i.e. $\mathrm{abs}^{f}_{I^{\mathtt{wthd},\mathtt{rthd}}_{\mathrm{S,RBL}},\mathrm{RBL}}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$ holds. We need to provide $\mathtt{stmp}'$, so', and $g : \langle E', \mathrm{po}', \mathtt{stmp}'\rangle.\mathrm{SEvent} \to \mathcal{G}.\mathrm{SEvent}$ respecting some conditions. From $\langle E', \mathrm{po}'\rangle$, we simply take $\mathtt{stmp}' = \mathtt{stmp}_{\mathrm{RBL}}$.

Since the implementation $I^{\mathtt{wthd},\mathtt{rthd}}_{\mathrm{S,RBL}}$ maps events that do not respect rthd or wthd to non-terminating loops, the abstraction f tells us that every event in E' does respect these functions.

Since $\mathcal{G}$ is {sv}-consistent, it means $(\mathsf{ppo} \cup \mathsf{so})^+ \subseteq \mathsf{hb}$, hb is transitive and irreflexive, and $\mathcal{G}$ is sv-consistent. Firstly, it means that for all thread t we have $\mathrm{po}|_t$ is a strict total order. From the properties of $\mathsf{abs}^{f}_{I^{\mathtt{wthd,rthd}}_{\mathtt{S,RBL}},\mathtt{RBL}}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$, we can easily see that it implies $\mathrm{po}'|_t$ is also a strict total order. Secondly, $\mathtt{stmp} = \mathtt{stmp}_{\mathtt{SV}}$ and there exists well-formed $\mathtt{v_R}$, $\mathtt{v_W}$, rf, and mo such that $[\mathtt{aCR}]; (\mathrm{po}^{-1} \cap \mathsf{rb}); [\mathtt{aCW}] = \emptyset$ and $\mathsf{so} = \mathsf{iso} \cup \mathsf{rf_e} \cup \mathsf{pf} \cup \mathsf{rb} \cup \mathsf{mo}$. Note that here mo is necessarily included in ppo and is not relevant by itself.

Let us define g.

- For $\mathrm{e}' = (t, \_, (\mathtt{Submit}^{\mathrm{RBL}}, (x, \widetilde{v}), \mathrm{true}))$, from the definition of the implementation and the abstraction f, there is some events $\mathrm{e}_w = (t, \_, (\mathtt{Write}_{\mathrm{sv}}, (h^x, v), ()))$ and $\mathrm{e}_b = (t, \_, (\mathtt{Bcast}_{\mathrm{sv}}, (h^x, d_x, s_n), ()))$ in $f^{-1}(\mathrm{e}')$, where $s_n = \{\mathtt{n}(t_i) \mid t_i \in \mathtt{rthd}(x)\} \setminus \{\mathtt{n}(t)\}$. We define $g(\mathrm{e}', \mathtt{aCW}) = (\mathrm{e}_w, \mathtt{aCW})$, and for every $n \in s_n$ we define $g(\mathrm{e}', \mathtt{aNRW}_n) = (\mathrm{e}_b, \mathtt{aNRW}_n)$.
- For $\mathrm{e}' = (t, \_, (\mathtt{Submit}^{\mathrm{RBL}}, (x, \widetilde{v}), \mathrm{false}))$, the first event of the implementation is $\mathrm{e}_r = (t, \_, (\mathtt{Read}_{\mathrm{sv}}, (h^x), v))$ and we define $g(\mathrm{e}', \mathtt{aWT}) = (\mathrm{e}_r, \mathtt{aCR})$.
- For $\mathrm{e}' = (t, \_, (\mathtt{Receive}^{\mathrm{RBL}}, (x), r))$, the second event of the implementation is of the form $\mathrm{e}_r = (t, \_, (\mathtt{Read}_{\mathrm{sv}}, (h^x), v))$. If the $\mathtt{Receive}^{\mathrm{RBL}}$ succeeds (i.e. $r = \widetilde{v}$) we define $g(\mathrm{e}', \mathtt{aCR}) = (\mathrm{e}_r, \mathtt{aCR})$. If the $\mathtt{Receive}^{\mathrm{RBL}}$ fails (i.e. $r = \bot$) we define $g(\mathrm{e}', \mathtt{aWT}) = (\mathrm{e}_r, \mathtt{aCR})$.

Since the stamps $\mathtt{aCR}$ and $\mathtt{aWT}$ have the same relations to other stamps (see Fig. 10), the first property of g holds.

For every event in E', we note $\mathtt{in}$ and $\mathtt{out}$ the values of the corresponding counter (h^x for a $\mathtt{Submit}^{\mathrm{RBL}}$, $h^x_{t_i}$ for a $\mathtt{Receive}^{\mathrm{RBL}}$) before and after the function call.

- For $\mathrm{e}' = (t, \_, (\mathtt{Submit}^{\mathrm{RBL}}, (x, \widetilde{v}), r)) \in E'$, there is some event $\mathrm{e}_r = (t, \_, (\mathtt{Read}_{\mathrm{sv}}, (h^x), v)) \in f^{-1}(\mathrm{e}')$. We define $\mathtt{in}(\mathrm{e}') = v$. If this function fails (i.e. $r = \mathtt{false}$), we define $\mathtt{out}(\mathrm{e}') = v$ as well. Otherwise, from the implementation there is $\mathrm{e}_w = (t, \_, (\mathtt{Write}_{\mathrm{sv}}, (h^x, v'), ())) \in f^{-1}(\mathrm{e}')$ and we define $\mathtt{out}(\mathrm{e}') = v'$.
- Similarly for $\mathrm{e}' = (t, \_, (\mathtt{Receive}^{\mathrm{RBL}}, (x), r)) \in E'$. There is $\mathrm{e}_r = (t, \_, (\mathtt{Read}_{\mathrm{sv}}, (h^x_t), v)) \in f^{-1}(\mathrm{e}')$, and in case of success there is $\mathrm{e}_w = (t, \_, (\mathtt{Write}_{\mathrm{sv}}, (h^x_t, v'), \_)) \in f^{-1}(\mathrm{e}')$. For a failure we have $\mathtt{in}(\mathrm{e}') = \mathtt{out}(\mathrm{e}') = v$, and for a success $\mathtt{in}(\mathrm{e}') = v$ and $\mathtt{out}(\mathrm{e}') = v'$.

We extend the notation to subevents: $\mathtt{in}((\mathrm{e}', a)) \triangleq \mathtt{in}(\mathrm{e}')$, and similarly for $\mathtt{out}$. We have some basic properties about $\mathtt{in}$ and $\mathtt{out}$.

- The first event of each thread has an $\mathtt{in}$ value of 0, and we always have $0 \leq \mathtt{in}(\mathrm{e}') \leq \mathtt{out}(\mathrm{e}')$
- For an event e' with label $(\mathtt{Submit}^{\mathrm{RBL}}, (x, (v_1, \ldots, v_V)), \mathtt{true})$, we have $\mathtt{out}(\mathrm{e}') = \mathtt{in}(\mathrm{e}') + V + 1$
- For an event e' with label $(\mathtt{Receive}^{\mathrm{RBL}}, (x), (v_1, \ldots, v_V))$, we have $\mathtt{out}(\mathrm{e}') = \mathtt{in}(\mathrm{e}') + V + 1$
- Let $E'|_{\mathtt{Submit}^{\mathrm{RBL}},x}$ be the subset of E' for calls to $\mathtt{Submit}^{\mathrm{RBL}}$ on x, and $\mathrm{po}'|_{\mathtt{Submit}^{\mathrm{RBL}},x}$ the corresponding subset of po'. If $(\mathrm{e}'_1, \mathrm{e}'_2) \in (\mathrm{po}'|_{\mathtt{Submit}^{\mathrm{RBL}},x})|_{\mathrm{imm}}$, then $\mathtt{out}(\mathrm{e}'_1) = \mathtt{in}(\mathrm{e}'_2)$. I.e., if we have two consecutive $\mathtt{Submit}^{\mathrm{RBL}}$ calls (e'_1 and e'_2) on x, the value of h^x at the end of the execution of e'_1 is equal to the value at the beginning of the execution of e'_2.
  This comes from the semantics of sv. A $\mathtt{Read}_{\mathrm{sv}}$ is required to read the last value written in program order. It cannot read from another thread or a broadcast as there is no other writing on h^x by definition of the implementation. It cannot read from a later write, since $[\mathtt{aCW}]; (\mathsf{rf} \cap \mathrm{po}^{-1}); [\mathtt{aCR}] \subseteq (\mathsf{rf_e} \cap \mathsf{ppo}^{-1}) \subseteq (\mathsf{hb} \cap \mathsf{hb}^{-1}) = \emptyset$. It cannot read from an earlier write than the last, since $[\mathtt{aCR}]; (\mathrm{po}^{-1} \cap \mathsf{rb}); [\mathtt{aCW}] = \emptyset$.
- Similarly for $\mathtt{Receive}^{\mathrm{RBL}}$, we can define $E'|_{\mathtt{Receive}^{\mathrm{RBL}},x}$ and $\mathrm{po}'|_{\mathtt{Receive}^{\mathrm{RBL}},x}$. If $(\mathrm{e}'_1, \mathrm{e}'_2) \in (\mathrm{po}'|_{\mathtt{Receive}^{\mathrm{RBL}},x})|_{\mathrm{imm}}$, then $\mathtt{out}(\mathrm{e}'_1) = \mathtt{in}(\mathrm{e}'_2)$ by the same reasoning.

We then choose the following relation rf'.

$$\mathsf{rf}' \triangleq \bigcup_{n,x} \mathsf{rf}'^{n}_{x} \qquad \mathsf{rf}'^{n}_{x} \triangleq (\mathcal{W}^{n}_{x} \times \mathcal{R}^{n}_{x}) \cap \left\{(\mathsf{s}'_1, \mathsf{s}'_2) \;\middle|\; \mathtt{in}(\mathsf{s}'_1) = \mathtt{in}(\mathsf{s}'_2)\right\}$$

We take $\mathsf{so}' = \mathsf{rf}' \cup \mathsf{fb}'$, where $\mathsf{fb}' \triangleq \bigcup_{n,x} \left(\mathcal{G}'.\mathcal{F}^{n}_{x} \times \mathcal{G}'.\mathcal{W}^{n}_{x} \setminus (\mathsf{po}'^{-1}; \mathsf{rf}'^{-1})\right)$. We need to prove that $g(\mathsf{so}') \subseteq \mathsf{hb}$ and that $\mathcal{G}' = \langle E', \mathsf{po}', \mathtt{stmp}', \mathsf{so}', \_\rangle$ is RBL-consistent. Since E' respects the functions `rthd` and `wthd`, we only need to check that rf' is well-formed for the latter.

As an intermediary result, let us show that if $(\mathsf{s}'_1, \mathsf{s}'_2) \in \mathsf{rf}'$, with $\mathsf{s}'_1 = (\mathsf{e}'_1, \mathtt{aNRW}_n)$ and $\mathsf{s}'_2 = (\mathsf{e}'_2, \mathtt{aCR})$ (i.e. they are on a location x with $\mathtt{in}(\mathsf{e}'_1) = \mathtt{in}(\mathsf{e}'_2)$), then the two events write/read the same tuple $\widetilde{v}$ and we have $\mathtt{out}(\mathsf{e}'_1) = \mathtt{out}(\mathsf{e}'_2)$. We have $\mathsf{e}'_1 = (t_1, \_, (\mathtt{Submit}^{\mathrm{RBL}}, (x, \widetilde{v}), \mathtt{true}))$ and $\mathsf{e}'_2 = (t_2, \_, (\mathtt{Receive}^{\mathrm{RBL}}, (x), \widetilde{v}'))$. Let us name $H = \mathtt{in}(\mathsf{e}'_1)$, $V = \mathtt{len}(\widetilde{v})$, and $V' = \mathtt{len}(\widetilde{v}')$. we aim to show that $\widetilde{v} = \widetilde{v}'$, which would also imply $\mathtt{out}(\mathsf{e}'_1) = \mathtt{in}(\mathsf{e}'_1) + V + 1 = \mathtt{in}(\mathsf{e}'_2) + V' + 1 = \mathtt{out}(\mathsf{e}'_2)$.

From the implementation of e'_1, we have $\mathsf{e}_{w_1} = (t_1, \_, (\mathtt{Write}_{\mathrm{sv}}, (h^x, H+V+1), ())) \in f^{-1}(\mathsf{e}'_1)$ and $\mathsf{e}_{b_1} = (t_1, \_, (\mathtt{Bcast}_{\mathrm{sv}}, (h^x, d_x, \_), ())) \in f^{-1}(\mathsf{e}'_1)$. We necessarily have $((\mathsf{e}_{w_1}, \mathtt{aCW}), (\mathsf{e}_{b_1}, \mathtt{aNLR}_n)) \in \mathsf{rf}$, and thus $\mathsf{v}_{\mathsf{W}}((\mathsf{e}_{b_1}, \mathtt{aNRW}_n)) = \mathsf{v}_{\mathsf{R}}((\mathsf{e}_{b_1}, \mathtt{aNLR}_n)) = H + V + 1 = \mathtt{out}(\mathsf{e}'_1)$. This is because e_{b_1} cannot read from an earlier read, as that would be ignoring e_{w_1} which is forbidden (from $\mathsf{rb} \in \mathsf{so}$), and cannot read from a later read because of the $\mathtt{Wait}_{\mathrm{sv}}(d_x)$ operation (from $\mathsf{pf} \in \mathsf{so}$) placed within each successful execution of $\mathtt{Submit}^{\mathrm{RBL}}(x, \_)$, making sure h^x is read by the broadcast before we can modify it again. This also holds for any broadcast on h^x of other events.

From the implementation of e'_2 we have $\mathsf{e}_r = (t_2, \_, (\mathtt{Read}_{\mathrm{sv}}, (h^x), H')) \in f^{-1}(\mathsf{e}'_2)$. From the inequality in the implementation we have $H' > H = \mathtt{in}(\mathsf{e}'_2)$, so $H' \neq 0$ and the value is read from a broadcast from thread t_1. There is $\mathsf{e}'_3 \in E'$ such that $\mathsf{e}_{b_3} = (t_1, \_, (\mathtt{Bcast}_{\mathrm{sv}}, (h^x, d_x, \_), ())) \in f^{-1}(\mathsf{e}'_3)$ with $\mathsf{v}_{\mathsf{W}}((\mathsf{e}_{b_3}, \mathtt{aNRW}_n)) = H' = \mathtt{out}(\mathsf{e}'_3)$ and $((\mathsf{e}_{b_3}, \mathtt{aNRW}_n), (\mathsf{e}_r, \mathtt{aCR})) \in \mathsf{rf}_{\mathsf{e}}$. We might have $\mathsf{e}'_1 = \mathsf{e}'_3$ and $\mathsf{e}_{b_1} = \mathsf{e}_{b_3}$, but not necessarily. Since $\mathtt{out}(\mathsf{e}'_3) = H' > H = \mathtt{in}(\mathsf{e}'_1)$, we necessarily have $(\mathsf{e}'_1, \mathsf{e}'_3) \in (\mathsf{po}')^*$ and thus by transitivity $((\mathsf{e}_{b_1}, \mathtt{aNRW}_n), (\mathsf{e}_r, \mathtt{aCR})) \in \mathsf{hb}$. Note that this can be written $(g(\mathsf{s}'_1), g(\mathsf{s}'_2)) \in \mathsf{hb}$, which proves $g(\mathsf{rf}') \subseteq \mathsf{hb}$.

The implementation of e'_1 makes several write and broadcast events of the form $\mathsf{e}_{w_i} = (t_1, \_, (\mathtt{Write}_{\mathrm{sv}}, (x_i, v_i), ())) \in f^{-1}(\mathsf{e}'_1)$ and $\mathsf{e}_{b_i} = (t_1, \_, (\mathtt{Bcast}_{\mathrm{sv}}, (x_i, \_, \_), ())) \in f^{-1}(\mathsf{e}'_1)$, with $i = (H + k)\%S$ for $0 \leq k \leq V$. Note: no location x_i is written twice, since $V + 1 \leq S$ from the condition in the implementation. Similarly, the implementation of e'_2 makes several read events of the form $\mathsf{e}_{r_i} = (t_2, \_, (\mathtt{Read}_{\mathrm{sv}}, (x_i), v'_i)) \in f^{-1}(\mathsf{e}'_2)$. It would be enough to check that each of these read event reads the value written by the corresponding write (i.e. $v_i = v'_i$).

Firstly, the value written is available. We have $(\mathsf{e}_{w_i}, \mathtt{aCW}) \xrightarrow{\mathsf{ppo}} (\mathsf{e}_{b_i}, \mathtt{aNLR}_n) \xrightarrow{\mathsf{iso}} (\mathsf{e}_{b_i}, \mathtt{aNRW}_n) \xrightarrow{\mathsf{ppo}} (\mathsf{e}_{b_1}, \mathtt{aNRW}_n) \xrightarrow{\mathsf{hb}} (\mathsf{e}_r, \mathtt{aCR}) \xrightarrow{\mathsf{ppo}} (\mathsf{e}_{r_i}, \mathtt{aCR})$, so since hb is transitive and irreflexive this implies $((\mathsf{e}_{b_i}, \mathtt{aNLR}_n), (\mathsf{e}_{w_i}, \mathtt{aCW})) \notin \mathsf{rb}$ and $((\mathsf{e}_{r_i}, \mathtt{aCR}), (\mathsf{e}_{b_i}, \mathtt{aNRW}_n)) \notin \mathsf{rb}$, and we cannot read from earlier values.

Secondly, we need to check that e_{b_i} and e_{r_i} cannot read from later values. Let us take $\mathsf{e}_{w'_i} = (t_1, \_, (\mathtt{Write}_{\mathrm{sv}}, (x_i, \_), ())) \in f^{-1}(\mathsf{e}'_3)$ and $\mathsf{e}_{b'_i} = (t_1, \_, (\mathtt{Bcast}_{\mathrm{sv}}, (x_i, \_, \_), ())) \in f^{-1}(\mathsf{e}'_3)$ from some later (in po') successful $\mathtt{Submit}^{\mathrm{RBL}}$ e'_3 on x. We use the index $\__3$ to indicate values of the execution of e'_3. We have i of the form $(H_3 + k_3)\%S$, for some $0 \leq k_3 \leq V_3$. Since $(\mathsf{e}'_1, \mathsf{e}'_3) \in \mathsf{po}'$ we have $H_3 = \mathtt{in}(\mathsf{e}'_3) \geq \mathtt{out}(\mathsf{e}'_1) > (H + k)$. Thus from $(H + k)\%S = i = (H_3 + k_3)\%S$ we have $H + k + S \leq H_3 + k_3 \leq H_3 + V_3$, i.e. the indices before modulo differ by at least the size S of the buffer. From the condition in the implementation of $\mathtt{Submit}^{\mathrm{RBL}}$, we have $(H_3 - M_3) + (V_3 + 1) \leq S$, and so $M_3 \geq H_3 + V_3 - S + 1 \geq H + k + 1 > H = \mathtt{in}(\mathsf{e}'_2)$. Intuitively, this large value of M_3 indicates that e'_2 is already finished. The implementation of e'_3 makes a read $\mathsf{e}_{r_3} = (t_1, \_, (\mathtt{Read}_{\mathrm{sv}}, (h^x_{t_2}), v_3))$ with $v_3 \geq M_3 > \mathtt{in}(\mathsf{e}'_2)$. The implementation of e'_2 makes a write $\mathsf{e}_{w_2} = (t_2, \_, (\mathtt{Write}_{\mathrm{sv}}, (h^x_{t_2}, \mathtt{out}(\mathsf{e}'_2)), ()))$.

By our properties of in and out, this is the first write on $h^x_{t_2}$ with value greater than H. Thus we have $((e_{w_2}, \mathsf{aCW}), (e_{r_3}, \mathsf{aCR})) \in \mathsf{hb}$ by ppo transitivity to the write being read, and via the intermediary of some broadcast. Since $((e_{r_i}, \mathsf{aCR}), (e_{w_2}, \mathsf{aCW})) \in \mathsf{ppo}$, $((e_{r_3}, \mathsf{aCR}), (e_{w'_i}, \mathsf{aCW}))) \in \mathsf{ppo}$ (thus $((e_{b_i}, \mathsf{aNLR}_n), (e_{w'_i}, \mathsf{aCW}))) \in \mathsf{hb}$), and $((e_{r_3}, \mathsf{aCR}), (e_{b'_i}, \mathsf{aNRW}_n))) \in \mathsf{ppo}$, we have that e_{r_i} cannot read from a later broadcast and e_{b_i} cannot read from a later write.

Thus $\widetilde{v} = \widetilde{v'}$ and $\mathtt{out}(e'_1) = \mathtt{out}(e'_2)$, which concludes our intermediary result. The same property holds for $((e'_1, \mathsf{aCW}), (e'_2, \mathsf{aCR})) \in \mathsf{rf}'$ for similar reasons. Except that if both threads are on the same node the reader can directly read the data without the help of broadcasts.

From this intermediary result, it is easy to check that rf' is well-formed.

- rf' is total and functional on its range. It is functional as two different successful $\mathtt{Submit}^{\mathrm{RBL}}$ events on x necessarily have different in values. We can check rf'^{-1} is total by contradiction, by taking the first (lowest in value) event $s'_r \in \mathcal{G}'.\mathcal{R}^n_x$ that is not related in rf'. If there is $s'_2 \in \mathcal{G}'.\mathcal{R}^n_x$ with $(s'_2, s'_r) \in (\mathsf{po}'|_{\mathtt{Receive}^{\mathrm{RBL}},x})|_{\mathsf{imm}}$, then by hypothesis there is s'_1 such that $(s'_1, s'_2) \in \mathsf{rf}'$. From our intermediary result we have $\mathtt{in}(s'_r) = \mathtt{out}(s'_2) = \mathtt{out}(s'_1)$. If there is a next successful $\mathtt{Submit}^{\mathrm{RBL}}$ event s'_w, then it would necessarily have $\mathtt{in}(s'_w) = \mathtt{out}(s'_1) = \mathtt{in}(s'_r)$, and thus we would have $(s'_w, s'_r) \in \mathsf{rf}'$, a contradiction. Such an event must exist because s'_r is successful: from the implementation, s'_r reads h^x and finds a value strictly higher than $\mathtt{out}(s'_1)$, which requires the existence of later $\mathtt{Submit}^{\mathrm{RBL}}$ events.
- Events related in rf' write and read the same tuple of values, from our intermediary result.
- Each thread can read each value once. This is because two successful $\mathtt{Receive}^{\mathrm{RBL}}$ calls on x from the same thread will have different in values and cannot read from the same $\mathtt{Submit}^{\mathrm{RBL}}$.
- Threads cannot jump a value. This is easily checked by induction. The first successful $\mathtt{Receive}^{\mathrm{RBL}}$ (with in value 0) must read the first successful $\mathtt{Submit}^{\mathrm{RBL}}$ (with in value 0). Whenever a successful $\mathtt{Receive}^{\mathrm{RBL}}$ occurs reading a specific $\mathtt{Submit}^{\mathrm{RBL}}$, the following successful $\mathtt{Submit}^{\mathrm{RBL}}$/$\mathtt{Receive}^{\mathrm{RBL}}$ events will have the same in value, and thus have to be related by rf'.

Finally, we are left to prove that $g(\mathsf{so}') \subseteq \mathsf{hb}$. During the proof of the intermediary result, we already checked $g(\mathsf{rf}') \subseteq \mathsf{hb}$. Let $(s'_f, s'_w) \in \mathsf{fb}'$, where $s'_f \in \mathcal{G}'.\mathcal{F}^n_x$ is a failed $\mathtt{Receive}^{\mathrm{RBL}}$ on x and $s'_w \in \mathcal{G}'.\mathcal{W}^n_x$ a successful $\mathtt{Submit}^{\mathrm{RBL}}$. We will assume they are on different nodes, $s'_f = (e'_f, \mathsf{aWT})$ and $s'_w = (e'_w, \mathsf{aNRW}_n)$, but the same reasoning can be adapted (without the broadcasts) to threads on the same node. Note that we necessarily have $\mathtt{in}(e'_f) \geq \mathtt{in}(e'_w)$. By contradiction, if we had $\mathtt{in}(e'_w) < \mathtt{in}(e'_f)$ then by induction and using our intermediary result we can see there is $s'_r \in \mathcal{G}'.\mathcal{R}^n_x$ such that $(s'_r, s'_f) \in \mathsf{po}'$ and $\mathtt{in}(s'_r) = \mathtt{in}(s'_w)$, thus $(s'_w, s'_r) \in \mathsf{rf}'$ contradicting $(s'_f, s'_w) \in \mathsf{fb}'$. The implementation of e'_f comprises two events: $e_1 = (t_f, \_, (\mathtt{Read}_{\mathrm{sv}}, (h^x_{t_f}), H))$ and $e_2 = (t_f, \_, (\mathtt{Read}_{\mathrm{sv}}, (h^x), H'))$ with $H' \leq H = \mathtt{in}(e'_f)$. The implementation of e'_w ends with two events: $e_3 = (t_w, \_, (\mathtt{Write}_{\mathrm{sv}}, (h^x, \mathtt{out}(e'_w)), ()))$ and $e_4 = (t_w, \_, (\mathtt{Bcast}_{\mathrm{sv}}, (h^x, d_x, \_), ()))$. We have $g(s'_f) = (e_2, \mathsf{aCR})$ and $g(s'_w) = (e_4, \mathsf{aNRW}_n)$, both events accessing the location h^x. As seen previously, we have $\mathtt{v_W}((e_4, \mathsf{aNRW}_n)) = \mathtt{out}(e'_w)$ as the broadcast can only read from e_3. We have $\mathtt{v_R}(g(s'_f)) = H' \leq \mathtt{in}(e'_f) \leq \mathtt{in}(e'_w) < \mathtt{out}(e'_w) = \mathtt{v_W}(g(s'_w))$. Since the value of h^x increases, we necessarily have $(g(s'_f), g(s'_w)) \in \mathsf{rb} \subseteq \mathsf{so} \subseteq \mathsf{hb}$. □

Corollary G.8. *The implementation* $I^{\mathtt{wthd,rthd}}_{\mathrm{S,RBL}}$ *is sound.*

G.5 RDMA$^{\text{WAIT}}$ to RDMA$^{\text{TSO}}$

Theorem G.9. *Let $\widetilde{\mathsf{p}}$ be a program using only the RDMA$^{\text{WAIT}}$ library. Then we have* $\mathtt{outcome}_{RDMA^{TSO}}(\llbracket\widetilde{\mathsf{p}}\rrbracket_{I_W}) \subseteq \mathtt{outcome}_{\{RDMA^{WAIT}\}}(\widetilde{\mathsf{p}})$.

Proof. By definition, we are given $\mathcal{G} = \langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}\rangle$ RDMA$^{\text{TSO}}$-consistent (Definition F.1) such that $\langle\widetilde{v}, \langle E, \mathrm{po}\rangle\rangle \in [\![\llbracket\widetilde{\mathsf{p}}\rrbracket_{I_W}]\!]$. Among others, it means $\langle E, \mathrm{po}\rangle$ respects nodes and there exists well-formed v_R, v_W, rf, mo, nfo, and pf such that ib is irreflexive, $\mathtt{stmp} = \mathtt{stmp}_{\mathsf{TSO}}$, so = iso $\cup$ rf$_e$ $\cup$ [aNLW]; pf $\cup$ nfo $\cup$ rb $\cup$ mo $\cup$ ([Inst]; ib), and hb $\triangleq$ (ppo $\cup$ so)$^+$ is irreflexive.

From Lemma E.2, since $\widetilde{\mathsf{p}}$ uses only RDMA$^{\text{WAIT}}$, there is E', po', f such that $\langle\widetilde{v}, \langle E', \mathrm{po}'\rangle\rangle \in [\![\widetilde{\mathsf{p}}]\!]$ and $\mathsf{abs}^{f}_{I_W,\text{RDMA}^{\text{WAIT}}}(\langle E, \mathrm{po}\rangle, \langle E', \mathrm{po}'\rangle)$. Note that this clearly implies $\langle E, \mathrm{po}\rangle$ also respects nodes, as the implementation I_W keeps the same locations. Our objective is to find stmp′, so′, and hb′ such that $\mathcal{G}' = \langle E', \mathrm{po}', \mathtt{stmp}', \mathrm{so}', \mathrm{hb}'\rangle$ is {RDMA$^{\text{WAIT}}$}-consistent (Definitions 3.6 and F.2). Of course, we pick $\mathtt{stmp}' \triangleq \mathtt{stmp}_{\mathsf{RL}}$ as it is the only choice for consistency. We will also pick hb′ $\triangleq$ (ppo′ $\cup$ so′)$^+$ since there is no external constraints. Thus, we only need to carefully pick so′ and show it works.

While our objective is not exactly local soundness (Definition 3.13), we still use a concretisation function $g : \langle E', \mathrm{po}', \mathtt{stmp}'\rangle.\mathrm{SEvent} \to \mathcal{G}.\mathrm{SEvent}$ to then define so′.

- For $\mathsf{e}' = (t, \_, (\mathtt{Write}, (x, v), ()))$, from the definition of the implementation I_W and the abstraction f, there is some event $\mathsf{e} = (t, \_, (\mathtt{Write}^{\text{TSO}}, (x, v), ())) \in f^{-1}(\mathsf{e}')$. We define $g(\mathsf{e}', \mathtt{aCW}) = (\mathsf{e}, \mathtt{aCW})$. For events calling Read, CAS, Mfence, and Rfence, we proceed similarly and let g map each subevent to their counterpart in the implementation.
- For $\mathsf{e}' = (t, \_, (\mathtt{Get}, (x, y, d), ()))$, there is some event $\mathsf{e} = (t, \_, (\mathtt{Get}^{\text{TSO}}, (x, y), (v))) \in f^{-1}(\mathsf{e}')$. We define $g(\mathsf{e}', \mathtt{aNRR}_{n(y)}) = (\mathsf{e}, \mathtt{aNRR}_{n(y)})$ and $g(\mathsf{e}', \mathtt{aNLW}_{n(y)}) = (\mathsf{e}, \mathtt{aNLW}_{n(y)})$. We proceed similarly for Put events.
- Finally for $\mathsf{e}' = (t, \_, (\mathtt{Wait}, (d), ()))$, there is in $f^{-1}(\mathsf{e}')$ some last event (in po order) of the form $\mathsf{e} = (t, \_, (\mathtt{SetIsEmpty}, (d^N), \mathtt{true}))$ confirming the set d^N tracking operations towards the last node N is empty. We define $g(\mathsf{e}', \mathtt{aWT}) = (\mathsf{e}, \mathtt{aMF})$.

We can see that $g(\langle \mathsf{e}', a'\rangle) = \langle \mathsf{e}, a\rangle$ implies that $f(\mathsf{e}) = \mathsf{e}'$ and that a is more restrictive than a'.

Each subevent in $\mathcal{G}'.\mathcal{R}$ (resp. $\mathcal{G}'.\mathcal{W}$) is mapped through g to a subevent in $\mathcal{G}.\mathcal{R}$ (resp. $\mathcal{G}.\mathcal{W}$) using the same stamp and location. Thus it is straightforward to define v'_R, v'_W, rf′, mo′, and nfo′ by relying on their counterparts in $\mathcal{G}$. E.g. $\mathsf{v}'_R(\mathsf{s}') \triangleq \mathsf{v}_R(g(\mathsf{s}'))$ and $\mathrm{rf}' \triangleq \{(\mathsf{s}'_1, \mathsf{s}'_2) \mid (g(\mathsf{s}'_1), g(\mathsf{s}'_2)) \in \mathrm{rf}\}$. The well-formedness of v_R, v_W, rf, mo, and nfo trivially implies that of v'_R, v'_W, rf′, mo′, and nfo′. From this, we have all the expected derived relations, including pfg′, pfp′, and ib′ $\triangleq$ (ippo′ $\cup$ iso′ $\cup$ rf′ $\cup$ pfg′ $\cup$ pfp′ $\cup$ nfo′ $\cup$ rb$'_{\mathsf{i}}$)$^+$. We then define so′ $\triangleq$ iso′ $\cup$ rf$'_{\mathsf{e}}$ $\cup$ pfg′ $\cup$ nfo′ $\cup$ rb′ $\cup$ mo′ $\cup$ ([Inst]; ib′), and as previously mentioned hb′ $\triangleq$ (ppo′ $\cup$ so′)$^+$.

To show {RDMA$^{\text{WAIT}}$}-consistency, we are left to prove that ib′ and hb′ are irreflexive. For this, it is enough to show that $g(\mathrm{ib}') \subseteq \mathrm{ib}$ and $g(\mathrm{hb}') \subseteq \mathrm{hb} \triangleq (\mathrm{ppo} \cup \mathrm{so})^+$ since we know both ib and hb to be irreflexive.

For all subevent s′, $g(\mathsf{s}')$ has a more restrictive stamp than s′ (in most cases it is the same stamp, but for Wait the stamp aMF is more restrictive than aWT); this implies that $g(\mathrm{ppo}') \subseteq \mathrm{ppo}$. Then, by definition, it is trivial to check that $g(\mathrm{rf}') \subseteq \mathrm{rf}$, $g(\mathrm{mo}') \subseteq \mathrm{mo}$, $g(\mathrm{nfo}') \subseteq \mathrm{nfo}$, $g(\mathrm{ippo}') \subseteq \mathrm{ippo}$, $g(\mathrm{rf}'_{\mathsf{e}}) \subseteq \mathrm{rf}_{\mathsf{e}}$, $g(\mathrm{iso}') \subseteq \mathrm{iso}$, $g(\mathrm{rb}') \subseteq \mathrm{rb}$, and $g(\mathrm{rb}'_{\mathsf{i}}) \subseteq \mathrm{rb}_{\mathsf{i}}$.

To finish the proof, we need the following crucial pieces: $g(\mathrm{pfp}') \subseteq \mathrm{ib}$, $g(\mathrm{pfg}') \subseteq \mathrm{ib}$, and $g(\mathrm{pfg}') \subseteq \mathrm{hb}$. In fact, it is enough to show that $g(\mathrm{pfp}')$ and $g(\mathrm{pfg}')$ are both included in pf; ppo$^+$. This is because pf; ppo$^+$ $\subseteq$ ib, [aNLW]; pf; ppo$^+$ $\subseteq$ hb, and the domain of $g(\mathrm{pfg}')$ is included in $\cup_n \mathcal{G}.\mathtt{aNLW}_n$ by definition.

Let $((\mathsf{e}'_1, \mathtt{aNRW}_n), (\mathsf{e}'_2, \mathtt{aWT})) \in \mathrm{pfp}'$. By definition they are of the form $\mathsf{e}'_1 = (t, \_, (\mathtt{Put}, (x, y, d), ()))$ and $\mathsf{e}'_2 = (t, \_, (\mathtt{Wait}, (d), ()))$, for some t, x, y, and d, with $(\mathsf{e}'_1, \mathsf{e}'_2) \in \mathrm{po}'$ and $n = \mathsf{n}(x)$ the remote

node of this operation. By definition of the implementation and the abstraction, $f^{-1}(\mathrm{e}'_1)$ contains two events $\mathrm{e}_1 = (t, \_, (\mathtt{Put}^{\mathrm{TSO}}, (x, y), (v)))$ and $\mathrm{e}_a = (t, \_, (\mathtt{SetAdd}, (d^n, v), ()))$, with $\mathrm{e}_1 \xrightarrow{\mathrm{po}} \mathrm{e}_a$. Meanwhile $f^{-1}(\mathrm{e}'_2)$ contains a last event $\mathrm{e}_2 = (t, \_, (\mathtt{SetIsEmpty}, (d^N), \mathtt{true}))$ and an earlier event $\mathrm{e}_3 = (t, \_, (\mathtt{SetIsEmpty}, (d^n), \mathtt{true}))$, with $\mathrm{e}_3 \xrightarrow{\mathrm{po}^*} \mathrm{e}_2$, confirming operations towards n are done (if $n = N$ then $\mathrm{e}_2 = \mathrm{e}_3$).

Since $f(\mathrm{e}_a) = \mathrm{e}'_1 \xrightarrow{\mathrm{po}'} \mathrm{e}'_2 = f(\mathrm{e}_3)$ and f is an abstraction, we have $\mathrm{e}_a \xrightarrow{\mathrm{po}} \mathrm{e}_3$, i.e. the value v is added to d^n before the moment d^n is confirmed empty. By consistency (Definition F.1), there is an in-between event $\mathrm{e}_4 = (t, \_, (\mathtt{SetRemove}, (d^n, v), ()))$ that removes this value, with $\mathrm{e}_a \xrightarrow{\mathrm{po}} \mathrm{e}_4 \xrightarrow{\mathrm{po}} \mathrm{e}_3$. From the definition of the implementation, such an event e_4 is immediately preceded (with maybe other `SetRemove` in-between) by an event $\mathrm{e}_p = (t, \_, (\mathtt{Poll}, (n), (v)))$. Now we argue that we necessarily have $((\mathrm{e}_1, \mathtt{aNRW}_n), (\mathrm{e}_p, \mathtt{aWT})) \in \mathrm{pf}$. From the well-formedness of pf, we know that $(\mathrm{e}_p, \mathtt{aWT})$ has a preimage (pf is total and functional on its range) and that this preimage outputs the value v. By consistency (Definition F.1), e_1 is the only $\mathtt{Get}^{\mathrm{TSO}}$ or $\mathtt{Put}^{\mathrm{TSO}}$ with output v. Thus $(\mathrm{e}_1, \mathtt{aNRW}_n)$ is the preimage of $(\mathrm{e}_p, \mathtt{aWT})$ by pf.

Finally we have $g(\mathrm{e}'_1, \mathtt{aNRW}_n) = (\mathrm{e}_1, \mathtt{aNRW}_n) \xrightarrow{\mathrm{pf}} (\mathrm{e}_p, \mathtt{aWT}) \xrightarrow{\mathrm{ppo}} (\mathrm{e}_4, \mathtt{aMF}) \xrightarrow{\mathrm{ppo}} (\mathrm{e}_3, \mathtt{aMF}) \xrightarrow{\mathrm{ppo}^*} (\mathrm{e}_2, \mathtt{aMF}) = g(\mathrm{e}'_2, \mathtt{aWT})$, which shows $g(\mathrm{pfp}') \subseteq \mathrm{pf}; \mathrm{ppo}^+$.

We similarly have $g(\mathrm{pfg}') \subseteq \mathrm{pf}; \mathrm{ppo}^+$ via the same reasoning. Thus ib' and hb' are irreflexive, and $\mathcal{G}'$ is $\{\textsc{rdma}^{\mathrm{WAIT}}\}$-consistent. □

G.6 Mixed-size writes Library

G.6.1 The MSW Library. A limitation of the $\textsc{rdma}^{\mathrm{WAIT}}$ library is that each location corresponds to a specific memory location, and thus can only contain a fixed amount of data. LOCO wants to provide abstractions simulating shared memory with distributed objects. As such, we want to hide away the atomicity constraints of the underlying RDMA technology and provide methods to manipulate large objects without the risk of wrong manipulations and corrupted data. A first step for this is the mixed-size write library (MSW) that can manipulate data of any size with the same semantics as $\textsc{rdma}^{\mathrm{WAIT}}$. The library uses similar methods, with a syntax defined as follows.

$$\begin{aligned} m(\widetilde{v}) ::= {} & \mathtt{Write}^{\mathrm{MSW}}(x, \langle v_1, \ldots, v_k \rangle) \mid \mathtt{TryRead}^{\mathrm{MSW}}(x) \\ & \mid \mathtt{Get}^{\mathrm{MSW}}(x, y, d) \mid \mathtt{Put}^{\mathrm{MSW}}(x, y, d) \mid \mathtt{Wait}^{\mathrm{MSW}}(d) \end{aligned}$$

There is two differences with the methods of the $\textsc{rdma}^{\mathrm{WAIT}}$ library. Firstly, the read function $\mathtt{TryRead}^{\mathrm{MSW}} : \mathsf{Loc} \to \mathsf{Val}^* \uplus \{\bot\}$ can fail if the underlying data is not in a stable state (i.e. corrupted or being modified). Secondly, the reads and writes $\mathtt{Write}^{\mathrm{MSW}} : \mathsf{Loc} \times \mathsf{Val}^* \to ()$ methods manipulate tuples of values, whereas $\textsc{rdma}^{\mathrm{WAIT}}$ locations can only hold a single value. If necessary, a more usual read method can be derived by simply looping calls to $\mathtt{TryRead}^{\mathrm{MSW}}$ until it succeeds.

The consistency predicate is then a copy of the one from $\textsc{rdma}^{\mathrm{WAIT}}$, except failing reads are ignored. This semantics guarantees there is no out-of-thin-air: if a $\mathtt{TryRead}^{\mathrm{MSW}}$ operation succeeds, then it reads a value that was explicitly written by some $\mathtt{Write}^{\mathrm{MSW}}$ operation.

This library can then be used to implement an MSW-Broadcast library where each shared variable contains a tuple of values, similarly to how SV is built on top of $\textsc{rdma}^{\mathrm{WAIT}}$.

Implementation. We assume given a function $\mathtt{size} : \mathsf{Loc} \to \mathbb{N}$ associating locations to the amount of data they hold. From this, we define the implementation $I^{\mathtt{size}}_{\mathrm{MSW}}$ of the MSW library into $\textsc{rdma}^{\mathrm{WAIT}}$. We assume some function hash, such that $\mathtt{hash}(\widetilde{v}) = \mathtt{hash}(\widetilde{v'})$ implies $\widetilde{v} = \widetilde{v'}$. For each location x of the MSW library, we create $\mathtt{size}(x) + 1$ locations $\{x_0, x_1, \ldots, x_{\mathtt{size}(x)}\}$ of the $\textsc{rdma}^{\mathrm{WAIT}}$ library. The location x_0 holds the hash of the data, which is written to $x_1, \ldots, x_{\mathtt{size}(x)}$.

$I^{\mathsf{size}}_{\mathsf{MSW}}(t, \mathtt{Write}^{\mathrm{MSW}}, (x, \langle v_1, \ldots, v_{\mathsf{size}(x)}\rangle)) \triangleq$
  $\mathtt{Write}(x_0, \mathsf{hash}((v_1, \ldots, v_{\mathsf{size}(x)})))$
  $\mathtt{Write}(x_1, v_1);$
  $\ldots;$
  $\mathtt{Write}(x_{\mathsf{size}(x)}, v_{\mathsf{size}(x)});$
$I^{\mathsf{size}}_{\mathsf{MSW}}(t, \mathtt{TryRead}^{\mathrm{MSW}}, (x)) \triangleq$
  $\mathtt{let}\ v_0 = \mathtt{Read}(x_0)\ \mathtt{in}$
  $\mathtt{let}\ v_1 = \mathtt{Read}(x_1)\ \mathtt{in}$
  $\ldots$
  $\mathtt{let}\ v_{\mathsf{size}(x)} = \mathtt{Read}(x_{\mathsf{size}(x)})\ \mathtt{in}$
  $\mathtt{if}\ v_0 = \mathsf{hash}(\langle v_1, \ldots, v_{\mathsf{size}(x)}\rangle)\ \mathtt{then}\ \langle v_1, \ldots, v_{\mathsf{size}(x)}\rangle\ \mathtt{else}\ \bot$

$I^{\mathsf{size}}_{\mathsf{MSW}}(t, \mathtt{Put}^{\mathrm{MSW}}, (x, y, d)) \triangleq$
  $\mathtt{Put}(x_0, y_0, d);$
  $\ldots;$
  $\mathtt{Put}(x_{\mathsf{size}(x)}, y_{\mathsf{size}(x)}, d));$
$I^{\mathsf{size}}_{\mathsf{MSW}}(t, \mathtt{Get}^{\mathrm{MSW}}, (x, y, d)) \triangleq$
  $\mathtt{Get}(x_0, y_0, d);$
  $\ldots;$
  $\mathtt{Get}(x_{\mathsf{size}(x)}, y_{\mathsf{size}(x)}, d));$
$I^{\mathsf{size}}_{\mathsf{MSW}}(t, \mathtt{Wait}^{\mathrm{MSW}}, (d)) \triangleq \mathtt{Wait}(d)$

Fig. 28. Implementation $I^{\mathsf{size}}_{\mathsf{MSW}}$ of the MSW library into RDMA$^{\mathrm{WAIT}}$

For events that do not respect `size` or the nodes, the implementation is simply an infinite loop, similarly to the previous implementations. Otherwise, as shown in Fig. 28, we apply the RDMA$^{\mathrm{WAIT}}$ methods to each location, and a read succeeds if the hash corresponds to the accompanying data.

Theorem G.10. *The implementation $I^{\mathsf{size}}_{\mathsf{MSW}}$ is locally sound.*

Proof. See Theorem G.3. □

G.6.2 Correctness. This appendix completes Appendix G.6.1 on the definition of RDMA$^{\mathrm{WAIT}}$. Our model assumes a size function $\mathtt{size} : \mathsf{Loc} \to \mathbb{N}$ associating each location to the amount of data it stores. As mentioned, we have the 5 methods:

$$m(\widetilde{v}) ::= \mathtt{Write}^{\mathrm{MSW}}(x, (v_1, \ldots, v_k)) \mid \mathtt{TryRead}^{\mathrm{MSW}}(x) \mid \mathtt{Get}^{\mathrm{MSW}}(x, y, d) \mid \mathtt{Put}^{\mathrm{MSW}}(x, y, d) \mid \mathtt{Wait}^{\mathrm{MSW}}(d)$$

- $\mathtt{Write}^{\mathrm{MSW}} : \mathsf{Loc} \times \mathsf{Val}^* \to ()$
- $\mathtt{TryRead}^{\mathrm{MSW}} : \mathsf{Loc} \to \mathsf{Val}^* \uplus \{\bot\}$
- $\mathtt{Get}^{\mathrm{MSW}} : \mathsf{Loc} \times \mathsf{Loc} \times \mathsf{Wid} \to ()$
- $\mathtt{Put}^{\mathrm{MSW}} : \mathsf{Loc} \times \mathsf{Loc} \times \mathsf{Wid} \to ()$
- $\mathtt{Wait}^{\mathrm{MSW}} : \mathsf{Wid} \to ()$

While this syntax does not include a TSO memory fence (similarly to BAL in 3.4), a program can use both this library and the memory fence from RDMA$^{\mathrm{WAIT}}$.

We also define `loc` as expected: $\mathtt{loc}(\mathtt{Write}^{\mathrm{MSW}}(x, v)) = \mathtt{loc}(\mathtt{TryRead}^{\mathrm{MSW}}(x)) = \{x\}$; $\mathtt{loc}(\mathtt{Get}^{\mathrm{MSW}}(x, y, d)) = \mathtt{loc}(\mathtt{Put}^{\mathrm{MSW}}(x, y, d)) = \{x; y\}$; and $\mathtt{loc}(\mathsf{e}) = \emptyset$ otherwise.

Consistency predicate. Given an execution $\mathcal{G} = \langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle$, we define consistency similarly to RDMA$^{\mathrm{WAIT}}$. The main difference is that the $\mathtt{TryRead}^{\mathrm{MSW}}$ function reading a location can fail without justification.

We define the only valid stamping function $\mathtt{stmp}_{\mathsf{MSW}}$ as follows:

- A succeeding $\mathtt{TryRead}^{\mathrm{MSW}}$ has stamp aCR: $\mathtt{stmp}_{\mathsf{MSW}}((\_, \_, (\mathtt{TryRead}^{\mathrm{MSW}}, \_, \widetilde{v}))) = \{\mathtt{aCR}\}$.
- A failing $\mathtt{TryRead}^{\mathrm{MSW}}$ has stamp aWT: $\mathtt{stmp}_{\mathsf{MSW}}((\_, \_, (\mathtt{TryRead}^{\mathrm{MSW}}, \_, \bot))) = \{\mathtt{aWT}\}$.
- Other events follow $\mathtt{stmp}_{\mathsf{RL}}$ (*cf.* §F.2): events calling $\mathtt{Write}^{\mathrm{MSW}}$, $\mathtt{Put}^{\mathrm{MSW}}$, $\mathtt{Get}^{\mathrm{MSW}}$, and $\mathtt{Wait}^{\mathrm{MSW}}$ have respectively stamps aCW, $\mathtt{aNRR}_n$ and $\mathtt{aNLW}_n$, $\mathtt{aNLR}_n$ and $\mathtt{aNRW}_n$, and aWT.

We mark failed read events with the stamp aWT to simplify the definition. This stamp has the same to relation as aCR (*cf.* 10), and is thus equivalent, but we do not need to change our definition of $\mathcal{G}.\mathcal{R}$ covering all events stamped aCR.

Definition G.11 (MSW-consistency). $\mathcal{G} = \langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle$ is MSW-consistent if:

- $\langle E, \mathrm{po}\rangle$ is well-formed (as in RDMA$^{\mathrm{WAIT}}$);

- E respects the function `size`. I.e., for all event with label $(\texttt{Write}^{\text{MSW}}, (x, (v_1, \ldots, v_k)), ())$ or $(\texttt{TryRead}^{\text{MSW}}, (x), (v_1, \ldots, v_k))$ we have $k = \texttt{size}(x)$, and for all event with label $(\texttt{Get}^{\text{MSW}}, (x, y, d), ())$ or $(\texttt{Put}^{\text{MSW}}, (x, y, d), ())$ we have $\texttt{size}(x) = \texttt{size}(y)$.
- $\texttt{stmp} = \texttt{stmp}_{\text{MSW}}$;
- there exists well-formed v_R, v_W, rf, mo, and nfo (defined as in RDMA$^{\text{WAIT}}$) such that ib is irreflexive and $\text{so} = \text{iso} \cup \text{rf}_e \cup \text{pfg} \cup \text{nfo} \cup \text{rb} \cup \text{mo} \cup ([\texttt{Inst}]; \text{ib})$.

Note. The components v_R, rf, and rb do not cover failed read events. This weak semantics does not guarantee any read will eventually succeed, as they are allowed to fail for any reason. It means synchronisation (e.g. barrier) do not force written data to be available.

This semantics only guarantees that there is no out-of-thin-air; i.e. if a read succeeds then it returns a value that was explicitly written.

A more complex semantics ensuring that properly written data is not corrupted would be interesting. It would require a proper notion of data races, and lead to a semantics much more complex than that of RDMA$^{\text{WAIT}}$.

Implementation. The implementation $I^{\texttt{size}}_{\text{MSW}}$ of the MSW library into RDMA$^{\text{WAIT}}$ is discussed in Appendix G.6.1.

H Correctness Proof of `kvstore`

The `kvstore` object described in §6.2 is linearisable [Herlihy and Wing 1990b], and we here provide an proof of safety. Note that our proof leverages both the composition of linearisability and the mutual exclusion property of our locks, their use are simplified by the composable nature of LOCO channels.

Updates to the indices are protected by an array of `ticket_lock`. When a node tries to insert or delete a key, it first acquires the lock with index `key%NUM_LOCKS`. It then looks the key up in its local index. In the case of an insertion, if the key does not yet exist, the node first writes the value to a free slot in its local data array with the valid bit unset, increments the counter corresponding to that slot, updates the checksum, and then broadcasts the value's location and counter to other nodes on a `ringbuffer` called the *tracker*. Each node monitors the set of other nodes' trackers with a dedicated thread, which applies requested updates to the local index and then acknowledges the message. The inserter waits until all nodes have acknowledged its message, meaning the location of the key is present in all indices, and then marks the entry valid and releases the lock. Deletion is the reverse under the lock; marking the entry invalid, then broadcasting the deletion, and removing the entry once all nodes have acknowledged it.

To update the value mapped to a key, a node takes the lock corresponding to that key and looks up its location in the local index. If it exists, it writes the new value to that location (retaining the counter and valid bit), updates the checksum, then releases the lock. This write is fenced, to ensure it is ordered with the subsequent lock release.

To retrieve the value mapped to a key, a node need not take a lock, but simply looks up the key in the local index, failing if it is not found, and reads the value and accompanying metadata from their location on the corresponding node. If the checksum is incorrect due to a torn update, it retries. If the valid bit is not set (indicating an incomplete insertion/deletion), or the counter mismatches (indicating a stale local index), the reader can safely return `EMPTY`.

H.1 Preliminaries

We choose linearisation points [Herlihy and Wing 1990b] for each modification operation as follows. A `write` linearises when the key, value, and checksum are fully placed on the host node. A `delete` linearises when the `valid` bit is unset (before all nodes have modified their local index and

acknowledged the deletion). An `insert` linearises when the `valid` bit is set (after all nodes have modified their local index and acknowledged the insertion).

The linearisation points of reads are determined retrospectively depending on the read value.

Investigation of the algorithm determines every read consists of two steps (possibly repeated). (1) A fetch from the local index to determine the node and address of the key's associated value. (2) A remote read to this location. The remote read can result in one of three possible scenarios.

(1) If the read contents match the associated counter and checksum and the valid bit is set, the read linearises at the point of the remote read's execution and returns the read value.
(2) If the read contents and the associated checksum do not match, the read overlaps with an ongoing (torn) update, and the read is retried in its entirety.
(3) If the read contents match the requested counter but the valid bit is unset, this implies either that an in-progress insert has not yet linearised, or an in-progress deletion has already linearised but not yet updated the local index. The read linearises at the point of the remote read's execution and returns `EMPTY`.
(4) If the read contents do not match the requested counter, this implies an in-progress `delete` has completed but had not yet updated the local index when the read was initiated, and later operations have reused the slot. In this case, the read linearises immediately after the delete and returns `EMPTY`.

H.2 Proof of Safety

Lemma H.1. *All* `write` *s,* `delete` *s, and* `insert` *s for a given key form a total modification order which respects the real-time ordering of the operations.*

Proof. By mutual exclusion on the per-key lock, each operation's effects are completed before any subsequent operation. □

Lemma H.2. *Every* `read` *returns a value consistent with the total modification order and which respects real-time ordering of the operations.*

Proof. We break our proof into three cases contingent on the result of the remote read. In the first, the local index counter matches the result of the remote read, in the second, the local index does not match, in the third, the checksum does not match and the read cannot determine the case. We validate the linearisation of the read for each case in reverse order.

In the case where the checksum does not match, this is an atomicity violation, and the operation retries without linearising.

In the case where the local index does not match, the counter value read by the remote read indicates that the local index is out of date. This case implies an in-progress `delete` has linearised but not yet updated the local index, and later operations have reused the slot. As the remote `delete` cannot complete until the local index is updated, the read must have overlapped in real-time with the delete, and thus can return `EMPTY`.

In the case where the local index matches, the remote read may discover a either a valid or invalid value. If the value is valid, the read can return the read value, as this value respects the most recent linearisation of a modification to the location. If the read discovers an invalid flag — this indicates that its local index is out-of-date with respect to an ongoing `delete` or `insert`. Returning `EMPTY` respects the linearisation point of both operations (note the asymmetry of the modifying operations to enable this possibility). □

By lemma H.1 and H.2, and by composition of linearisable objects [Herlihy and Wing 1990b],

Theorem H.3. *The presented hashmap is linearisable.*