# 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.

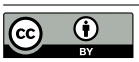
CCS Concepts: • **Theory of computation → Axiomatic semantics**; **Distributed computing models**; • **Software and its engineering → Distributed programming languages**.

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), 32 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 [1990], 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 [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$^{\text{TSO}}$ (which combines an RDMA networking fabric with Intel x86-TSO nodes), making use of an existing formalisation by Ambal et al. [2024]. RDMA$^{\text{TSO}}$ 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$^{\text{WAIT}}$, a thin layer of abstraction over RDMA$^{\text{TSO}}$, 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$^{\text{TSO}}$ model.

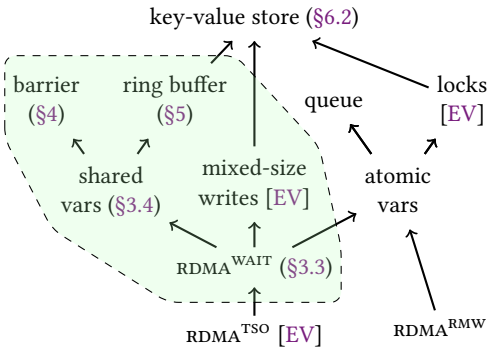


Fig. 1. LOCO libraries with their dependencies

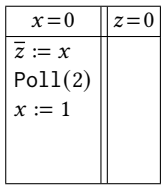
***MOWGLI.*** As a second contribution, we introduce Mowgli (MOdular Weak Graph-based LIbraries), a generic, modular framework for modelling and verifying weak libraries. Mowgli is *generic* in that it makes no assumptions about the underlying memory model (e.g. RDMA or TSO) in its core theory; and it 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$^{\text{WAIT}}$ 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$^{\text{TSO}}$ model. The verified libraries are highlighted in Fig. 1.

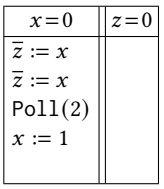
Mowgli represents program executions as graphs, whose nodes are called *events* and represent either a simple operation like a read or a write, or a more complex operation such as a method call. Following the declarative approach of Raad et al. [2019] and Stefanesco et al. [2024], we specify each concurrent 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.

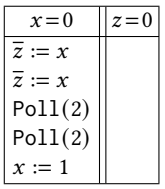
Our main result supporting modular proofs in Mowgli is a new locality result that decomposes proving correctness of a system into proofs about the correctness of its individual components. This is akin to the notion of compositionality for linearisability [Herlihy and Wing 1990], but generalised to a 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




(a) $z=0$ ✓ $z=1$ ✗

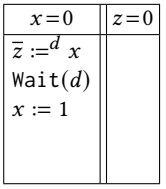



(b) $z=0$ ✓ $z=1$ ✓

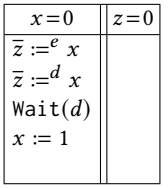



(c) $z=0$ ✓ $z=1$ ✗

Fig. 2. Polling under RDMA$^{\text{TSO}}$




(a) $z=0$ ✓ $z=1$ ✗




(b) $z=0$ ✓ $z=1$ ✗

Fig. 3. Waiting under RDMA$^{\text{WAIT}}$

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$^{\text{WAIT}}$, 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$^{\text{WAIT}}$ implementation over the existing RDMA$^{\text{TSO}}$ 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$^{\text{WAIT}}$, 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$^{\text{WAIT}}$ Memory Model

We start by informally describing LOCO's base memory model, RDMA$^{\text{WAIT}}$, and contrast it to RDMA$^{\text{TSO}}$ [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$^{\text{TSO}}$, 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 the extended version [EV].

## 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, in extended version).*** 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 the extended version [EV].

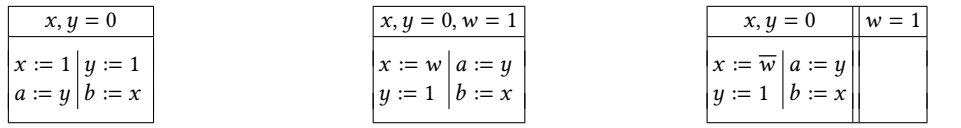
***Additional LOCO Libraries.*** In addition to the proven libraries that are the focus of this paper, LOCO [] [Hodgkins et al. 2025] contains a number of additional objects that rely on an RDMA read-modify-write primitive currently missing from the formalisation provided in RDMA$^{TSO}$. These include 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 [EV], and a distributed DC/DC converter [EV]. 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

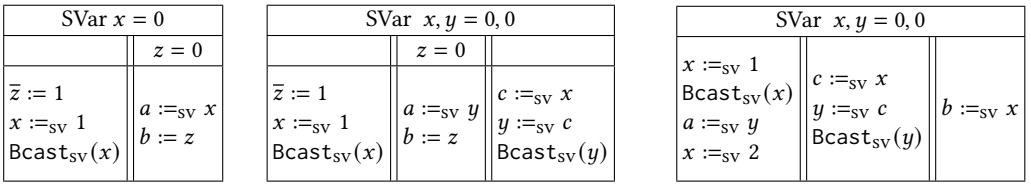
To support reasoning about LOCO libraries, we develop a modular verification framework for RDMA$^{WAIT}$ programs. Our point of departure is the Yacovet framework [Raad et al. 2019; Stefanesco et al. 2024] that was used to reason about weak shared memory *within* a single node. Yacovet, however, is not 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 $\mathsf{aNLR}_2$); **(2)** writing $x$ on node 2 (stamp $\mathsf{aNRW}_2$); **(3)** reading $x$ to send to node 3 (stamp $\mathsf{aNLR}_3$); **(4)** writing $x$ on node 3 (stamp $\mathsf{aNRW}_3$). With this we can create a more fine-grain sequence of dependencies: $x :=_{\mathrm{sv}} 1 \longrightarrow \langle \mathsf{Bcast}_{\mathrm{sv}}(x), \mathsf{aNLR}_2 \rangle \longrightarrow \langle \mathsf{Bcast}_{\mathrm{sv}}(x), \mathsf{aNRW}_2 \rangle \longrightarrow c :=_{\mathrm{sv}} x \longrightarrow \ldots \longrightarrow x :=_{\mathrm{sv}} 2 \longrightarrow \langle \mathsf{Bcast}_{\mathrm{sv}}(x), \mathsf{aNLR}_3 \rangle \longrightarrow \langle \mathsf{Bcast}_{\mathrm{sv}}(x), \mathsf{aNRW}_3 \rangle \longrightarrow b :=_{\mathrm{sv}} x$. 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 $\langle \overline{z} := 1, \mathsf{aNRW}_2 \rangle \xrightarrow{\text{ppo}} \langle \mathsf{Bcast}_{\mathrm{sv}}(x), \mathsf{aNRW}_2 \rangle$ 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. $\langle \mathsf{Bcast}_{\mathrm{sv}}(x), \mathsf{aNLR}_2 \rangle \longrightarrow \langle \overline{z} := 1, \mathsf{aNRW}_2 \rangle \longrightarrow \langle \mathsf{Bcast}_{\mathrm{sv}}(x), \mathsf{aNRW}_2 \rangle$, 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 $\iota$ 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:

$$\text{ppo} \triangleq \{\langle\langle e_1, a_1\rangle, \langle e_2, a_2\rangle\rangle \mid \langle e_1, e_2\rangle \in \text{po} \wedge a_1 \in \texttt{stmp}(e_1) \wedge a_2 \in \texttt{stmp}(e_2) \wedge \langle a_1, a_2\rangle \in \text{to}\}$$

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 $\texttt{stmp}_{\text{SV}}$:

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

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 $\texttt{Bcast}_{\text{sv}}(x, d, \{\ldots, n, \ldots\}); \texttt{GF}_{\text{sv}}(\{\ldots, n, \ldots\})$, the plain execution has two events $e_{BR}$ and $e_{GF}$, and the definitions of $\texttt{stmp}_{\text{SV}}$ and to (cell G11 in Fig. 10) imply $\langle e_{BR}, \texttt{aNRW}_n\rangle \xrightarrow{\text{ppo}} \langle e_{GF}, \texttt{aGF}_n\rangle$.

### 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 $\Lambda$ to denote a set of pairwise compatible libraries.

*Definition 3.6.* Let $\Lambda$ be a set of pairwise compatible libraries. An execution $\langle E, \mathrm{po}, \mathtt{stmp}, \mathrm{so}, \mathrm{hb}\rangle$ is $\Lambda$-*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 $\Lambda$-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 $\Lambda$, we note $\mathtt{outcome}_\Lambda(\widetilde{\mathsf{p}})$ the set of all output values of its $\Lambda$-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 $\mathsf{n}(x)$. From this, given $\langle E, \mathrm{po}\rangle$, there is a single valid stamping function $\mathtt{stmp}_{\mathsf{RL}}$. Notably we have $\mathtt{stmp}_{\mathsf{RL}}(\mathtt{Get}(x, y, d)) = \left\{\mathsf{aNRR}_{\mathsf{n}(y)}, \mathsf{aNLW}_{\mathsf{n}(y)}\right\}$ and $\mathtt{stmp}_{\mathsf{RL}}(\mathtt{Put}(x, y, d)) = \left\{\mathsf{aNLR}_{\mathsf{n}(x)}, \mathsf{aNRW}_{\mathsf{n}(x)}\right\}$. 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 $\{\mathsf{aMF}, \mathsf{aCR}\}$, as it behaves as both a memory fence (`aMF`) and a CPU read (`aCR`).

The formal semantics requires several functions and relations: $\mathsf{v}_\mathsf{R}$, $\mathsf{v}_\mathsf{W}$, 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 $\mathtt{loc}(\mathsf{e}) = \emptyset$ 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 $\{\mathsf{s} \in \mathcal{G}.\mathrm{SEvent} \mid \mathsf{s}.a = a\}$. For families, by abuse of notation, we also write e.g. $\mathcal{G}.\mathsf{aNRR}$ for $\bigcup_{n \in \mathrm{Node}} \mathcal{G}.\mathsf{aNRR}_n$. We extend the notation `loc` to subevents by writing $\mathtt{loc}(\mathsf{s})$ for $\mathtt{loc}(\mathsf{s.e})$. We define the set of *reads* as $\mathcal{G}.\mathcal{R} \triangleq \mathcal{G}.\mathsf{aCR} \cup \mathcal{G}.\mathsf{aCAS} \cup \mathcal{G}.\mathsf{aNLR} \cup \mathcal{G}.\mathsf{aNRR}$ and *writes* as $\mathcal{G}.\mathcal{W} \triangleq \mathcal{G}.\mathsf{aCW} \cup \mathcal{G}.\mathsf{aCAS} \cup \mathcal{G}.\mathsf{aNLW} \cup \mathcal{G}.\mathsf{aNRW}$. We write $\mathcal{G}.\mathcal{W}_x \triangleq \{\mathsf{s} \in \mathcal{G}.\mathcal{W} \mid \mathtt{loc}(\mathsf{s}) = \{x\}\}$ 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 $\mathrm{rf}^n$ 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 $\mathrm{mo}^n_x$ is a strict total order on $\mathcal{G}.\mathcal{W}^n_x$.

We further define the *reads-from-internal* relation as $\mathrm{rf}_\mathsf{i} \triangleq [\mathtt{aCW}]; (\mathrm{po} \cap \mathrm{rf}); [\mathtt{aCR}]$ (which corresponds to CPU reads and writes using the same TSO store buffer), and the *reads-from-external* relation as $\mathrm{rf}_\mathsf{e} \triangleq \mathrm{rf} \setminus \mathrm{rf}_\mathsf{i}$. As we shall see in Def. 3.8, $\mathrm{rf}_\mathsf{i}$ does *not* contribute to synchronisation order, whereas $\mathrm{rf}_\mathsf{e}$ 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 $\mathsf{v}_\mathsf{R}$, $\mathsf{v}_\mathsf{W}$, rf, and mo, such that $[\mathtt{aCR}]; (\mathrm{po}^{-1} \cap \mathrm{rb}); [\mathtt{aCW}] = \emptyset$ and $\mathrm{so} = \mathrm{iso} \cup \mathrm{rf}_\mathsf{e} \cup \mathrm{pf} \cup \mathrm{rb} \cup \mathrm{mo}$.

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 $\Lambda$ 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 $\Lambda$.
(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.

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

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

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

As an example, we can define the implementation $I_{\mathsf{SV}}$ of the broadcast library into RDMA$^{\mathrm{WAIT}}$. 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 RDMA$^{\mathrm{WAIT}}$ 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 $I_{\mathsf{SV}}$ 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 $\Lambda$ 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 $\Lambda$. 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 $\Lambda$, and that $G = \langle E, \mathrm{po}\rangle$ and $G' = \langle E', \mathrm{po}'\rangle$ are plain executions using methods of $\Lambda$ 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 $\mathsf{e}'$, 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 $\Lambda$-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 $\mathtt{stmp}'$, $\mathrm{so}'$, 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 $\mathsf{hb}'$ 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 $\Lambda$-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: $\mathsf{to}$ (via stamps) and $\mathsf{so}'$. If we have $\langle \mathrm{s}'_1, \mathrm{s}'_2\rangle \in \mathsf{so}'$ in the abstraction, then we require that the concretisation of $\mathrm{s}'_1$ synchronises with the concretisation of $\mathrm{s}'_2$, 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 $\mathsf{to}$ 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 $\mathsf{hb}$ is potentially bigger than $(\mathsf{ppo} \cup \mathsf{so})^+$. In which case, we need to prove the result for any reasonable $\mathsf{hb}'$ bigger than $(\mathsf{ppo}' \cup \mathsf{so}')^+$. Thus local soundness states that if the implementation has a $\Lambda$-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 the extended version [EV]. □

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

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

Proof. See the extended version [EV]. □

## 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 $\mathsf{e}_1$ and $\mathsf{e}_2$ *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 the extended version [EV]. □

### 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 $\mathtt{stmp}_{\mathsf{BAL}}(\langle \_, \_, \langle \mathsf{BAR}_{\mathrm{BAL}}, (x), () \rangle \rangle) = \bigcup_{n \in \mathrm{Node}} \{\mathsf{aGF}_n\} \cup \{\mathsf{aCR}\}$ and for the implementation, we define $I^{\mathsf{b}}_{\mathsf{BAL}}(t, \mathsf{BAR}_{\mathrm{BAL}}, (x)) \triangleq \mathtt{let}\, s_n = \mathrm{Node}\ \mathtt{in}\ \ldots$. 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 $\mathtt{Submit}^{\mathrm{RBL}} : \mathrm{Loc} \times \mathrm{Val}^* \to \mathbb{B}$ and $\mathtt{Receive}^{\mathrm{RBL}} : \mathrm{Loc} \to \mathrm{Val}^* \uplus \{\bot\}$, 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 $\mathtt{wthd} : \mathrm{Loc} \to \mathrm{Tid}$ and $\mathtt{rthd} : \mathrm{Loc} \to \mathcal{P}(\mathrm{Tid})$ 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) $\mathrm{rf}_x^n$ 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 $\mathsf{t}(\mathsf{s}_2) \neq \mathsf{t}(\mathsf{s}_3)$, 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 $\mathsf{s}_3$ 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 $\mathsf{s}_1$ is not available when $\mathsf{s}_3$ is executed. Either there is $\mathsf{s}_2$ 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 $\mathsf{s}_2$ 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.

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

$\texttt{For } t \notin \texttt{rthd}(x):$
$I^{\texttt{wthd},\texttt{rthd}}_{\text{S,RBL}}(t, \texttt{Receive}^{\text{RBL}}, (x)) \triangleq \texttt{loop } \{()\}$

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

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

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

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

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 $\text{hb}^{-1} \cap \text{fb} = \emptyset$. 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 $\texttt{Receive}^{\text{RBL}}$ has to succeed. With the weaker semantics, they are allowed to both fail, even when both $\texttt{Submit}^{\text{RBL}}$ calls succeed.

## 5.2 LOCO Implementation

As before, we assume given the functions $\texttt{wthd} : \text{Loc} \to \text{Tid}$ and $\texttt{rthd} : \text{Loc} \to \mathcal{P}(\text{Tid})$. 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 $\texttt{rthd}(x) = \{t_1, \ldots, t_k\}$ 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. $\texttt{loop } \{()\}$), similarly to other implementations. Otherwise, we use the implementation $I^{\texttt{wthd},\texttt{rthd}}_{\text{S,RBL}}$ 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 $\texttt{Submit}^{\text{RBL}}$, 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 the extended version [EV]. □

## 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 the extended version [EV].

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.

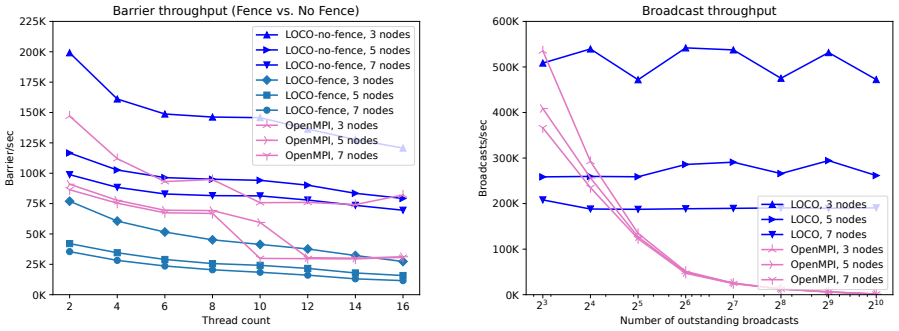



Fig. 15. Comparison of barrier and broadcast operations for LOCO and OpenMPI.

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).

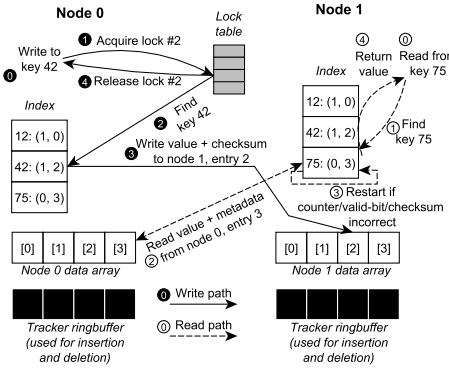



Fig. 16. `kvstore` read and write operations

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 the extended version [EV] – 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.]].

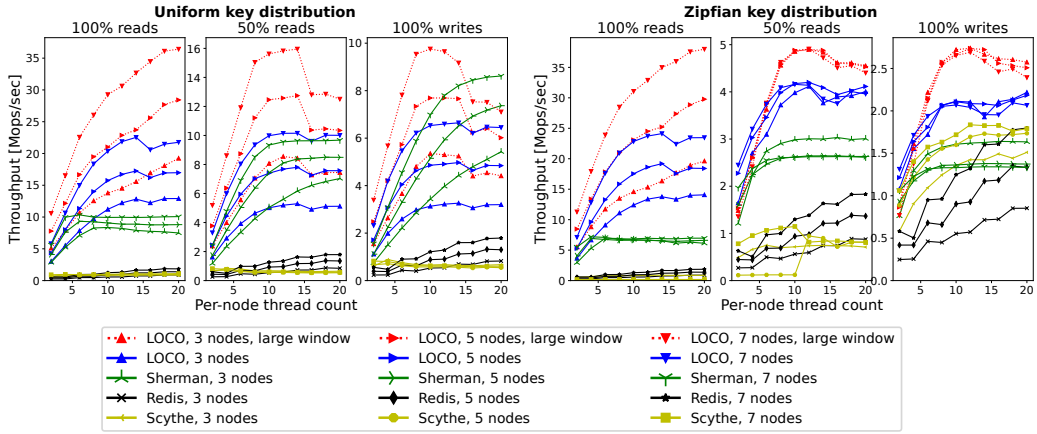



Fig. 17. Throughput comparison of key-value stores.

We modified Sherman to issue a fence ($\texttt{GF}_{\texttt{sv}}$) 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 the extended version [EV]. 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.]. 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.

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

Guillaume Ambal, George Hodgkins, Mark Madler, Gregory Chockler, Brijesh Dongol, Joseph Izraelevitz, Azalea Raad, and Viktor Vafeiadis. 2025. A Verified High-Performance Composable Object Library for Remote Direct Memory Access (Extended Version). (2025). arXiv:cs.PL/2510.10531 https://arxiv.org/abs/2510.10531

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

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

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

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

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.

Received 2025-07-10; accepted 2025-11-06