Specifying and Verifying RDMA Synchronisation

Abstract. Remote direct memory access (RDMA) allows a machine to directly read from and write to the memory of remote machine, enabling high-throughput, low-latency data transfer. Ensuring correctness of RDMA programs has only recently become possible with the formalisation of RDMA^{TSO} semantics (describing the behaviour of RDMA networking over a TSO CPU). However, this semantics currently lacks a formalisation of remote synchronisation, meaning that the implementations of common abstractions such as locks cannot be verified. In this paper, we close this gap by presenting $\text{RDMA}^{\text{TSO}}_{\text{RMW}}$, the first semantics for remote 'read-modify-write' (RMW) instructions over TSO. It turns out that remote RMW operations are weak and only ensure atomicity against other remote RMWs. We therefore build a set of composable synchronisation abstractions starting with the $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ library. Underpinned by $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$, we then specify, implement and verify three classes of remote locks that are suitable for different scenarios. Additionally, we develop the notion of a strong RDMA model, $\text{RDMA}^{\text{SC}}_{\text{RMW}}$, which is akin to sequential consistency in shared memory architectures. Our libraries are built to be compatible with an existing set of high-performance libraries called LOCO, which ensures compositionality and verifiability.

1 Introduction

Remote Direct Memory Access (RDMA), as implemented by RoCE and Infiniband, is a high-performance networking technology that enables low-latency wire-speed data transmission. Specifically, an RDMA device can directly read and write from the memory of a remote (network) *node* (machine), bypassing the remote CPU and operating system. RDMA technology has been used in high-performance computing applications (including supercomputers) since the early 2000s, and is being branched out to support a much wider range of applications, ranging from production-grade data centres [25, 32, 34] to distributed AI training [17]. Thus, there is currently a push towards developing programmer-friendly libraries to improve the reliability and robustness of such applications.

To enable rigorous development and verification, there is ongoing work aimed at formalising the semantics of RDMA architectures, primarily the RDMA memory model. Dan et al [13] proposed an early model, called coreRMA, which was used to formalise the behaviours of remote read/write operations, assuming a sequentially consistent CPU. Ambal et al [3] have presented a more realistic RDMA^{TSO} specification, which assumes a total-store-order (TSO) CPU (e.g. as implemented by Intel processors) that (unlike coreRMA) has been validated against real RoCE and Infiniband hardware. RDMA^{TSO} precisely describes the interaction between the CPU and NIC (Network Interface Card) and the reorderings that they allow. Their formalisation comprises both declarative and operational models (which are proved equivalent). However, RDMA^{TSO} only covers a *subset*

of RDMA instructions. In particular it only covers local (i.e. CPU-level) 'read-modify-write' (RMW) synchronisation, relegating remote (i.e. RDMA) RMWs to future work. This means that RDMA^{TSO} cannot be used to specify and verify locks and other related high-level mechanisms that require synchronisation at the network level.

In this work, we address this gap and extend the existing efforts with a notion of remote (RDMA) synchronisation. Specifically, we develop the $\text{RDMA}^{\text{TSO}}_{\text{RMW}}$ model by extending RDMA^{TSO} to account for remote RMWs. To ensure the fidelity of our extension, we developed $\text{RDMA}^{\text{TSO}}_{\text{RMW}}$ by careful inspection of the Infiniband technical manual [21] and in close consultation with engineers at NVIDIA, the largest manufacturer of RDMA products worldwide (after acquiring Mellanox in 2019). We then build a series of synchronisation libraries and prove them correct (as we discuss below). An overview of our development is given in Fig. 1.

Remote RMW instructions are surprisingly *weak* in that they only guarantee a weak form of isolation: remote RMWs are atomic *only* with respect to other remote RMWs and not CPU accesses or remote read and write accesses operations (*cf.* weak transactional isolation [8,10,16,28,29]). We provide a set of litmus tests that exemplify these behaviours in two- and three-node configurations. A second challenge is that (like RDMA^{TSO}) $\text{RDMA}^{\text{TSO}}_{\text{RMW}}$ is *not* compositional: the semantics of a certain remote operation, `Poll`, directly depends on the *exact number of remote operations* in the program up to that point! As such, one cannot specify the behaviour of `Poll` *modularly* (in isolation).

To address both issues, we build on the *Library of Composable Objects* (LOCO) framework [4, 20], which is a modular set of objects for constructing RDMA libraries. We start at the lowest level of LOCO, called $\text{RDMA}^{\text{WAIT}}$, which is a compositional analogue of RDMA^{TSO} (i.e. also does not support remote RMWs). As shown in Fig. 1, the $\text{RDMA}^{\text{WAIT}}$ library itself is implemented using RDMA^{TSO}.

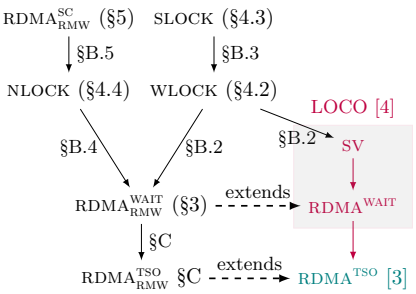


Fig. 1: Development overview

Importantly, $\text{RDMA}^{\text{WAIT}}$ abstracts RDMA^{TSO} by replacing its non-modular operation (`Poll`) with a modular analogue (`Wait`, see §2.1). As such, unlike RDMA^{TSO}, $\text{RDMA}^{\text{WAIT}}$ is *modular* and can be *composed* with other LOCO libraries (thanks to its `Wait` operation). Accordingly, we develop $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ by extending $\text{RDMA}^{\text{WAIT}}$ with RMW operations. Specifically, in $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ we specify two remote RMWs: `RCAS` (remote compare-and-swap) and `RFAA` (remote fetch-and-add). In doing so, we also ensure that our extensions are compatible with $\text{RDMA}^{\text{WAIT}}$ and the modular design of LOCO, thus guaranteeing that $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ is also modular and can be composed with other LOCO libraries.

We next use $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ to develop several RDMA libraries (Fig. 1). First, we combine $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ with the *shared variable* (SV) library (that provides a mechanism for broadcasting to many nodes) of LOCO to develop three lock libraries with varying synchronisation guarantees (§4), each offering a different trade-

off between intuitive behaviours and efficiency. Second, we develop an RDMA library with strong *sequential consistency* (SC) [23] semantics (§5).

Our first lock library is a *weak lock*, WLOCK, that provides *mutual exclusion* across multiple threads over the network, but does not provide any ordering guarantees on RDMA instructions enclosed within critical sections. Nevertheless, it is possible to recover such strong ordering guarantees on RDMA operations within a WLOCK critical section by inserting a *global fence* immediately before the lock is released. To capture this, we thus develop a *strong lock*, SLOCK, that guarantees the desired strong guarantees by executing a global fence before releasing the lock. The most novel aspect of our library is the notion of a *node lock*, NLOCK, that takes a node n as a parameter, and only guarantees synchronisation on RDMA operations specific to n, while operations within a critical section acting on a different node $n' \neq n$ are left unsynchronised.

Interestingly, we show that it is possible to build a novel, strong model for RDMA using NLOCK. Specifically, we develop the $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ library, which, unlike $\text{RDMA}^{\text{WAIT}}$, provides support for strong isolation of remote RMW instructions, with strong synchronisation akin to SC.[1]

For each library L in our development (Fig. 1), we 1) *formally specify* L; 2) develop a *reference implementation* of L using lower-level libraries; and 3) *prove* our implementation is *correct* against its specification. For (1) and (3), we use MOWGLI [4], a declarative framework previously used to verify a subset of LOCO (those *without* RMWs). MOWGLI is a compositional framework for specification and verification of very weak libraries where program order is not preserved (e.g. RDMA programs). However, previous definitions [4] are not sufficient to specify remote RMWs out of the box, and we extend them with the features needed (§3).

Contributions. Our core contributions are as follows. **(1)** We develop the *first formal semantics of remote RMWs* through the $\text{RDMA}^{\text{TSO}}_{\text{RMW}}$ and $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ models by carefully inspecting the (informal) technical specification [21]. Our models have further been validated by NVIDIA engineers. **(2)** We *generalise* MOWGLI to add the intricate features needed for specifying and verifying very weak libraries. We then use LOCO and our extension of MOWGLI to develop *several programmer-friendly and composable RDMA libraries*. Specifically, **(3)** we specify, implement and verify *three lock libraries* offering varying degrees of synchronisation guarantees and efficiency; and **(4)** we develop a novel, strong RDMA model, $\text{RDMA}^{\text{SC}}_{\text{RMW}}$, ensuring strong isolation of RDMA instructions with strong synchronisation guarantees of SC.

Outline. The remainder of this article is organised as follows. In §2 we discuss the necessary background and present an intuitive overview of our contributions. In §3 we describe how we extend the MOWGLI framework and present our $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ model. In §4 we present our three lock libraries (including their specification, implementation, and verification), which we build on top of $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$. In §5 we specify, implement, and verify our $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ library (simulating SC in RDMA programs). Finally, we discuss related work in §6.

[1] In related work, Ambal et al. [5] write RDMA^{SC} for an RDMA model where the underlying CPU is SC (instead of TSO). This is unrelated to $\text{RDMA}^{\text{SC}}_{\text{RMW}}$.

2 Background and Overview

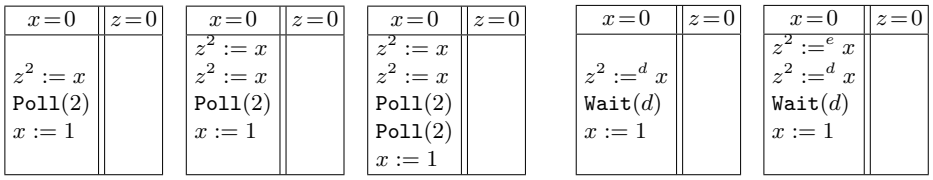
We present an intuitive account of our contributions via a series of litmus tests. We begin with a summary of necessary background (§2.1 and §2.2). We discuss the behaviour of remote RMW ('read-modify-write') synchronisation, culminating in our formal $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ model (§2.3). We then describe our RDMA libraries (§2.4), including locks and a library for sequential consistency we build from it.

Terminology and Litmus Test Notation. Throughout this article, we present small examples (litmus tests) to highlight particular behaviours. A single vertical bar (e.g. in Fig. 8a) separates threads on the *same* (network) node, while a double vertical bar (e.g. in Fig. 2a) separates *distinct* nodes. For each annotated outcome, ✓ denotes that the outcome is *allowed* by the semantics, while ✗ states that the outcome is *disallowed*. To distinguish local and remote (memory) locations, we write x^n for a location on a remote node n, and write x for a location on the current local node. We number nodes from left to right, starting at 1. The statement on the top line of each column denotes where locations reside as well as their initial values; e.g. $x = 0$ and $z = 0$ on top of Fig. 2a denote that x and z respectively reside on nodes 1 and 2 with initial value 0. When a thread on local node n issues a remote operation to be executed on remote node n', we denote this by stating that the operation is by n *towards* n'.

2.1 Background: RDMA^{TSO}, $\text{RDMA}^{\text{WAIT}}$, and LOCO

The RDMA^{TSO} Model. Ambal et al. [3] developed RDMA^{TSO}, the first formal model of RDMA programs where the underlying CPUs are assumed to follow the x86-TSO memory model [26]. RDMA^{TSO} formalises the semantics of *RDMA Writes* (referred to as *puts*), *RDMA Reads* (referred to as *gets*) and *polling* instructions, executed by the *network interface card* (NIC). A put operation towards n, written $x^n := y$, reads from local location y (referred to as a *NIC local read*) and writes to remote location x on node n (a *NIC remote write*). Similarly, a get operation towards n, written $y := x^n$ reads from remote location x (a *NIC remote read*) and writes to local location y (a *NIC local write*). The RDMA^{TSO} semantics is unintuitive as remote operations are executed by NIC *independently* from later CPU operations, *as if* run in parallel to them. For instance, the program $z^2 := x; x := 1$ (comprising a put towards node 2, followed by a standard CPU store) can result in z containing value 1 as follows: 1) CPU offloads the put instruction to the NIC; 2) CPU executes $x := 1$; 3) NIC executes the put, fetching the *new* value 1 of x and updating the remote location z in node 2 to this new value. To prevent this weak behaviour, a programmer can *poll* the remote instruction (towards node 2) by executing `Poll`(2), as shown in Fig. 2a: this blocks the CPU until the NIC confirms that the put has been executed, thereby preventing the above scenario.

The polling system on RDMA hardware (and thus RDMA^{TSO}) is highly brittle in that it synchronises with the *earliest* (in program order) unpolled remote operation. For instance, in Fig. 2b the single poll only acknowledges the first



(a) $z = 0$ ✓ $z = 1$ ✗ (b) $z = 0$ ✓ $z = 1$ ✓ (c) $z = 0$ ✓ $z = 1$ ✗

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

(a) $z = 0$ ✓ $z = 1$ ✗ (b) $z = 0$ ✓ $z = 1$ ✗

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

put, and the second put can be arbitrarily delayed, once again enabling the outcome $z = 1$. Preventing unintended weak behaviours therefore often relies on *counting* remote operations and polling them accordingly; e.g. in this case we must use two polls to prevent the weak outcome, as in Fig. 2c.

The RDMA$^{\text{WAIT}}$ Model. The non-local semantics of polls does not lend itself to compositional programming and verification. That is, the polling semantics depends on the *exact number* of earlier remote operations towards the same node. To address this, recent work developed LOCO [4] as an RDMA library for composable objects with a more abstract completion system that ensures modularity and compositionality through a *waiting* instruction that is analogous to polling but is compositional. Specifically, in LOCO each remote operation is associated with a *work identifier*, $d \in \mathsf{Wid}$, and the wait operation $\texttt{Wait}(d)$ ensures the completion of all previous operations with this identifier (multiple remote operations may have the same identifier). This is illustrated in Figs. 3a and 3b (obtained from Figs. 2a and 2b by replacing polls with waits), where $z^2 :=^d x$ denotes a put (as before) with work id d. Note that unlike in Fig. 2b, adding an earlier put in Fig. 3b (with different work id e) does not alter the behaviour of $\texttt{Wait}(d)$ and the weak outcome $z=1$ remains prohibited.

From a reordering perspective, RDMA$^{\text{WAIT}}$ is still quite permissive. For example, because a remote NIC sends an acknowledgement for a put as soon as it is received (but before the put takes effect in memory), RDMA$^{\text{WAIT}}$ permits the store-buffering behaviour in Fig. 4. Therefore, using RDMA$^{\text{WAIT}}$, LOCO additionally implements a *global-fence operation* towards a node n, written $\texttt{GFence}(\{n\})$, that blocks until all previous remote operations towards n are *fully* completed (see §4.1). Replacing $\texttt{Wait}(d)$ and $\texttt{Wait}(e)$ in Fig. 4 respectively with fences $\texttt{GFence}(\{2\})$ and $\texttt{GFence}(\{1\})$ would prevent the store-buffering behaviour.

2.2 Background: MOWGLI

To support compositional specification and verification, Ambal et al. have developed the MOWGLI framework [4]. They have specified the RDMA$^{\text{WAIT}}$ formal model (obtained from RDMA$^{\text{TSO}}$ by replacing the `poll` instruction with `Wait`) in MOWGLI and subsequently used it as a foundation for developing and verifying a suite of RDMA libraries. The principal one is a *shared variable* (SV) library (see §4.1), where each node possesses a local copy of each variable x. The methods include store ($x :=_{\text{SV}} v$) and load ($a :=_{\text{SV}} x$) operations to access the local

$y = 0$	$x = 0$
$x^2 :=^d 1$ $\texttt{Wait}(d)$ $a := y$	$y^1 :=^e 1$ $\texttt{Wait}(e)$ $b := x$

$(a, b) = (0, 0)$ ✓

Fig. 4: Store buffering

SVar $x = 0$	
	$z = 0$
$z^2 := 1$ $x :=_{\text{SV}} 1$ $\texttt{Bcast}_{\text{SV}}(x)$	$a :=_{\text{SV}} x$ $b := z$

(a) $(a, b) = (1, 0)$ ✗

SVar $x = 0$		
	$y, z = 0, 0$	
$z^2 := 1$ $x :=_{\text{SV}} 1$ $\texttt{Bcast}_{\text{SV}}(x)$	$a := y$ $b := z$	$c :=_{\text{SV}} x$ $y^2 := 1$

(b) $(a, b, c) = (1, 0, 1)$ ✓

Fig. 5: Shared variable examples

copy, as well as a broadcast ($\texttt{Bcast}_{\text{SV}}(x)$) operation to forward the local value to other nodes.

Specification. MOWGLI [4] is a declarative framework for modularly specifying and verifying libraries in the context of (very) weak concurrency models. Unlike other declarative frameworks in the literature [27, 31], MOWGLI can handle the behaviours allowed by RDMA programs. The key novelty in MOWGLI enabling this is the use of a fixed set of *stamps*, $\mathsf{Stamp} = \{a_1, \ldots\}$, and the *stamp-order* relation, $\mathsf{sto} \subseteq \mathsf{Stamp} \times \mathsf{Stamp}$, defined as a subset of the program order that is *preserved*. This then allows one to define weak libraries where the program order is not fully preserved, as is the case in RDMA.

We present the stamps and their ordering in Fig. 9 (assuming that the underlying CPUs follow the TSO model). Intuitively, each stamp denotes a behaviour category, such as a CPU write ($\texttt{aCW}$), a CPU read ($\texttt{aCR}$), a NIC remote read ($\texttt{aNRR}_n$) or write ($\texttt{aNRW}_n$) towards n, or a NIC local read ($\texttt{aNLR}_n$) or write ($\texttt{aNLW}_n$) towards n. Compared to [4], we also introduce a new stamp $\texttt{aNAR}_n$ to represent the ordering guarantees of remote RMWs (see §2.3).

This stamp mechanism addresses two problems. The first is the reordering of methods of different libraries. As libraries are defined *independently*, the exact interaction between pairs of methods of different libraries cannot be explicit. Instead, libraries can associate their method calls with generic behaviour categories (stamps), so that their interactions can be implicitly deduced. For instance, in Fig. 5a, $z^2 := 1$ and $b := z$ are part of $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$, while $\texttt{Bcast}_{\text{SV}}(x)$ and $a :=_{\text{SV}} x$ are part of the SV library. To determine if the outcome $(a, b) = (1, 0)$ is allowed, we need to check if $z^2 := 1$ and $\texttt{Bcast}_{\text{SV}}(x)$ can be reordered on node 1, and if $a :=_{\text{SV}} x$ and $b := z$ can be reordered on node 2. The semantics of the two libraries (§3 and §A) ensure that $z^2 := 1$ and $\texttt{Bcast}_{\text{SV}}(x)$ behave as remote writes towards node 2 (stamp $\texttt{aNRW}_2$) and that $a :=_{\text{SV}} x$ and $b := z$ behave as CPU reads (stamp $\texttt{aCR}$). This enforces their respective program orders as $\langle z^2 := 1, \texttt{aNRW}_2\rangle \xrightarrow{\mathsf{ppo}} \langle \texttt{Bcast}_{\text{SV}}(x), \texttt{aNRW}_2\rangle$ and $\langle a :=_{\text{SV}} x, \texttt{aCR}\rangle \xrightarrow{\mathsf{ppo}} \langle b := z, \texttt{aCR}\rangle$, where ppo is the *preserved program order*, i.e. they cannot be reordered. Moreover, if $a = 1$, then we have the *happens-before* (hb) relation $\langle \texttt{Bcast}_{\text{SV}}(x), \texttt{aNRW}_2\rangle \xrightarrow{\mathsf{hb}} \langle a :=_{\text{SV}} x, \texttt{aCR}\rangle$, and as $\mathsf{ppo} \subseteq \mathsf{hb}$, by transitivity we have $\langle z^2 := 1, \texttt{aNRW}_2\rangle \xrightarrow{\mathsf{hb}} \langle b := z, \texttt{aCR}\rangle$, i.e. the weak outcome $(a, b) = (1, 0)$ is prohibited.

The second problem stamps address is the *partial* execution of methods. A method call may have multiple visible effects, and observing one does not necessarily imply that others are also observed. In Fig. 5b the shared variable

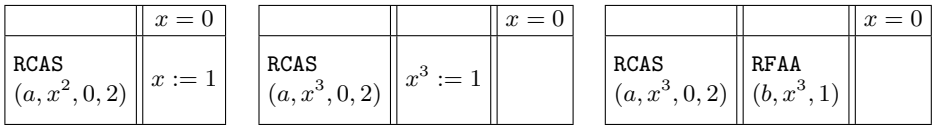
x is read by the *third* node, which then sends a message to node 2 (through $y^2 := 1$). As such, when $(a, c) = (1, 1)$, we have a $\xrightarrow{\mathsf{hb}}$ chain from $\mathtt{Bcast}_{\text{SV}}(x)$ to $b := z$ and may naturally expect $c = 1$. However, this is *not* case. Specifically, as per the semantics of SV, $\mathtt{Bcast}_{\text{SV}}(x)$ is associated with (at least) two stamps, $\mathtt{aNRW}_2$ (remote write towards node 2) and $\mathtt{aNRW}_3$ (remote write towards node 3), where the latter is observed but is *not* ordered with the earlier $z^2 := 1$ operation (as they are toward different nodes). That is, we have the hb orders $\langle z^2 := 1, \mathtt{aNRW}_2\rangle \xrightarrow{\mathsf{ppo} \subseteq \mathsf{hb}} \langle \mathtt{Bcast}_{\text{SV}}(x), \mathtt{aNRW}_2\rangle$ (as in example Fig. 5a) and $\langle \mathtt{Bcast}_{\text{SV}}(x), \mathtt{aNRW}_3\rangle \xrightarrow{\mathsf{hb}} \langle b := z, \mathtt{aCR}\rangle$, and when put together they do *not* imply $\langle z^2 := 1, \mathtt{aNRW}_2\rangle \xrightarrow{\mathsf{hb}} \langle b := z, \mathtt{aCR}\rangle$, allowing the weak outcome $b = 0$. In other words, $z^2 := 1$ and $\mathtt{Bcast}_{\text{SV}}(x)$ can be *partially* reordered: although their respective updates (on z and x) towards node 2 stay ordered, the update on x towards *other* nodes (i.e. node 3) may take place before $z^2 := 1$ is executed. Associating a method call with multiple stamps allows us to express such nuances.

Implementation and Soundness. Within the MOWGLI framework, Ambal et al. [4] also formalise the notion of a *library implementation* and what it means for an implementation I to be *sound* against its specification, i.e. that the behaviours of the implementation are contained in those of its specification. To enable proving implementation soundness *compositionally*, they establish a *local soundness* theorem. Specifically, to show that an implementation I of library L is correct, one must show that for *all* client programs P with calls to L (where P may in general contain calls to libraries other than L), replacing the calls to L with their corresponding (inlined) implementation yields the same outcome. Intuitively, as the only calls being replaced (inlined) are those of L, the calls to libraries other than L should not affect the outcome. That is, it should be sufficient to show that the implementation is *locally sound* by considering client programs that only constitute calls to L. Ambal et al. then prove that local soundness implies soundness: if I is a locally sound implementation of L (i.e. for all client programs that only comprise calls to L), then I is a sound implementation of L (i.e. for all client programs).

As we discuss below, we use MOWGLI to specify several RDMA libraries and verify their implementations, as shown in Fig. 1.

2.3 Remote Read-Modify-Write Operations

CPU Read-Modify-Writes. Read-modify-writes (RMW) are a category of synchronisation operations that simultaneously read the value v of a location and update (modify-write) it in place. Examples of common RMWs include the compare-and-swap, $\mathtt{CAS}(x, v_1, v_2)$, instruction (it reads the current value v of x and updates it to v_2 if $v = v_1$ and otherwise leaves it unchanged); and the fetch-and-add, $\mathtt{FAA}(x, v)$, instruction (it increments the value of x by v unconditionally). Both operations return the old value of x. These operations are useful for ensuring inter-thread synchronisation and are often used to implement strong synchronisation mechanisms such as locks (mutexes).



(a) $x = 2$ ✓ (b) $x = 2$ ✓ (c) $x = 2$ ✗

Fig. 6: Examples showcasing the limited atomicity of remote RMW operations

$x = 0$	
$a :=$ CAS$(x, 0, 2)$	$x := 1$

$x = 2$ ✗

CPU RMWs behave *atomically*: their 'read' and 'modify-write' phases cannot be interleaved by concurrent instructions. As such, RMWs are commonly referred to as 'atomic operations'. This is illustrated in the example across where the outcome $x = 2$ is disallowed. If the right thread executes first, then x is updated to 1 and subsequently the CAS fails. If the left thread executes first, then the right thread overwrites x to 1.

Remote RMWs. The RDMA hardware specification [21] optionally supports two remote RMW instructions, referred to as 'atomics[2]': RCAS(a, x, v_1, v_2), analogous to $a :=$ CAS(x, v_1, v_2) on CPUs, and RFAA(a, x, v), analogous to $a :=$ FAA(x, v) on CPUs, where x is a remote location in both cases.

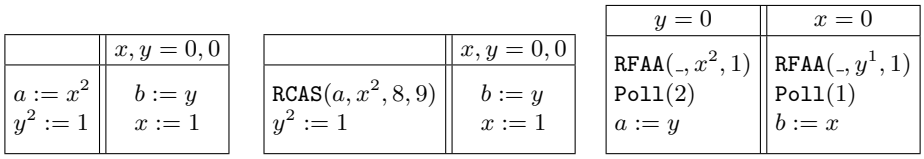
Unlike CPU RMWs, remote RMWs do *not* always behave atomically: their 'read' and 'modify-write' phases may be interleaved by other CPU or (remote) put/get instructions. This is illustrated in the examples of Figs. 6a and 6b, executing a remote CAS in parallel with a CPU store (Fig. 6a) and a put (Fig. 6b), where the remote CAS can first read 0 from x, be interleaved with the concurrent CPU store/put writing 1 to x, and then update x to 2.

This weakness is due to an inherent hardware limitation. Atomicity is possible on CPUs because a CPU core can: 1) request exclusive access to a cache line; 2) read the cache line; 3) write to the cache line; 4) release the cache line. During periods of exclusive access, other components (e.g. other CPU cores or the NIC) cannot access the cache line in-between the 'read' and 'modify-write'. This, however, is not feasible over RDMA since NICs cannot lock a cache line; they can only submit read and write operations to their PCIe root complex. As such, it is not possible to block accesses by other components (e.g. the CPU) interleaving between the NIC's 'read' and the 'modify-write'.

Nevertheless, remote RMWs do behave atomically with respect to other remote RMWs. For instance, as shown in Fig. 6c, a remote FAA cannot interleave between the 'read' and 'modify-write' phases of a remote CAS.

In practice, one can ensure atomicity of accesses to a location x by ensuring x is *only* ever accessed through remote RMWs. As such, it is common for RDMA programs to access *local* locations (i.e. those residing on their node) through remote RMWs (via loop-back). To provide atomicity between remote RMWs and other operations, we require software solutions, as supported by $\text{RDMA}^{\text{SC}}_{\text{RMW}}$.

[2] Although RMWs are commonly referred to as 'atomics' in the RDMA specification, they do *not* always behave atomically.



(a) $(a, b) = (1, 1)$ ✓ (b) $(a, b) = (1, 1)$ ✗ (c) $(a, b) = (0, 0)$ ✓

Fig. 7: Examples of remote RMW behaviours and how they compare from `Put`s.

Extending RDMA$^{\text{TSO}}$ with RMWs. The RDMA$^{\text{TSO}}$ and RDMA$^{\text{WAIT}}$ models do not include the semantics of remote RMWs; we close this gap in this work. Specifically, starting from RDMA$^{\text{TSO}}$ [3], we formulate RDMA$^{\text{TSO}}_{\text{RMW}}$ both declaratively and operationally and prove the two characterisations are equivalent (see §D).

Our main reference for modelling the semantics of remote RMWs is the Infiniband technical specification [21]. However, as the specification is often ambiguous, we developed our model in close collaboration with NVIDIA experts specialising in RDMA hardware who confirmed the expected behaviours of RMWs and that our model captures them faithfully.

Compared to RDMA$^{\text{TSO}}$, our RDMA$^{\text{TSO}}_{\text{RMW}}$ declarative model brings two important changes. The first is a new relation, the '*remote-atomic-order*' rao, capturing the mutual exclusion of remote RMWs. We require a total order rao$_n$ on remote RMWs towards each node n, such that rao$_n \subseteq$ hb (i.e. it induces synchronisation). The second is a new stamp, `aNAR`$_n$ (the 'NIC atomic read'), encoding the new ordering guarantees of the read phase of a remote RMW (see Fig. 9). Recall that a `Get` performs a NIC remote read (stamp `aNRR`$_n$) followed by a NIC local write (`aNLW`$_n$), while a `Put` performs a NIC local read (`aNLR`$_n$) followed by a NIC remote write (`aNRW`$_n$). Analogously, a remote RMW, e.g. RFAA(x, y, v), performs (up to) three NIC accesses: 1) it remotely reads y (`aNAR`$_n$); 2) remotely updates y (`aNRW`$_n$); and 3) locally writes (the return value) to x (`aNLW`$_n$).

Note that the stamp `aNAR`$_n$ is required because a remote read stamp (`aNRR`$_n$) is insufficient for modelling the stronger ordering guarantees of the 'read' phase of an RMW. We show an example of this in Figs. 7a and 7b. The (remote) read phase of a `Get` (`aNRR`$_n$) may be delayed (reordered) after a later remote write (`aNRW`$_n$ of a `Put` or RMW). As such, the weak 'load-buffering' behaviour in Fig. 7a is allowed. By contrast, the read phase of a remote RMW (`aNAR`$_n$) cannot be delayed, and thus the analogous behaviour is prohibited in Fig. 7b.

Finally, the example in Fig. 7c shows that the remote write ('modify') phase (`aNRW`$_n$) of an RMW behaves similarly to that of a `Put`. In particular, a poll does *not* enforce the full completion of the remote write and thus the weak 'store-buffering' behaviour presented is allowed, similarly to Fig. 4.

Supporting Modularity with RDMA$^{\text{WAIT}}_{\text{RMW}}$. As RDMA$^{\text{TSO}}_{\text{RMW}}$ is not modular, we develop RDMA$^{\text{WAIT}}_{\text{RMW}}$ by adapting RDMA$^{\text{WAIT}}$ [4, 20]. We then implement RDMA$^{\text{WAIT}}_{\text{RMW}}$ using RDMA$^{\text{TSO}}_{\text{RMW}}$ and prove that it is correct (§C) against its specification.

2.4 Modular RDMA Synchronisation Libraries

We now implement RDMA libraries *modularly* and specify and verify them using MOWGLI. A key use case of our remote RMWs is for implementing network-wide locks that ensure mutual exclusion of critical sections. A lock library provides two main operations, `Acq`(l) and `Rel`(l), for acquiring and releasing a lock l, respectively. When specifying such a network lock, there are several choices for defining its semantics as there are trade-offs between the guarantees (strength) of a lock and the efficiency of its implementation.

Fig. 8 presents several variants of an example where two threads use a lock l to access locations x and y in a critical section (the first thread writing to x and y and the last thread reading from x and y). As the locks are expected to ensure atomicity of the critical sections (enclosed within the lock acquisition and release blocks), the expected outcomes are either $a = b = 0$ or $a = b = 1$, i.e. not $a \neq b$. However, ensuring this strong guarantee for locks is not straightforward over RDMA. Specifically, while ensuring mutual exclusion is necessary for prohibiting the weak $a \neq b$ behaviour, it is not sufficient. We must additionally ensure that the operations enclosed in a critical section are *completed* before the end of the critical section (and hence are not *reordered* past the lock release). However, as we demonstrated above, meeting these latter constraints are not always straightforward due to the weak ordering guarantees on remote operations.

Weak Lock Library. The weakest network lock that we consider ensures mutual exclusion *only*, but does not prohibit the enclosed operations from being reordered. As shown in Fig. 8a, when the operations enclosed in a critical section are CPU loads and stores, the weak outcome $a \neq b$ is prohibited. By contrast, as shown in Fig. 8b, when the enclosed operations are over RDMA (two `Puts` in Fig. 8b), then a weak lock is insufficient and we may observe $a \neq b$. This is because the remote operations may not complete before the critical section ends.

Thus, we require an operation akin to a global fence (see §2.1) to ensure that these remote operations are completed. Note that as shown in Fig. 8c, the global fence in isolation (without the protection provided by a weak lock) is also insufficient for prohibiting the weak behaviour as it only provides intra-thread synchronisation (and does not ensure mutual exclusion). However, as shown in Fig. 8d, if we combine a weak lock with a global fence, we can attain the desired strong guarantees and prohibit $a \neq b$.

Strong Lock Library. The weak lock library discussed above is efficient and gives programmers full control over synchronisation. However, if not used correctly, without the relevant global fences, its guarantees are not as strong as one may expect. That is, in designing the weak lock library, we opted for better performance over the strength of guarantees. We next develop a *strong* lock library that achieves the desired strong guarantees (without the need for additional synchronisation via fences). Specifically, on releasing a strong lock *all* earlier operations are guaranteed to have *fully* completed.

This is illustrated in Fig. 8e, where the outcome $a \neq b$ is once again prohibited. However, the strong guarantees of strong locks come at the cost of their implementation efficiency. Intuitively, an implementation of a strong lock release

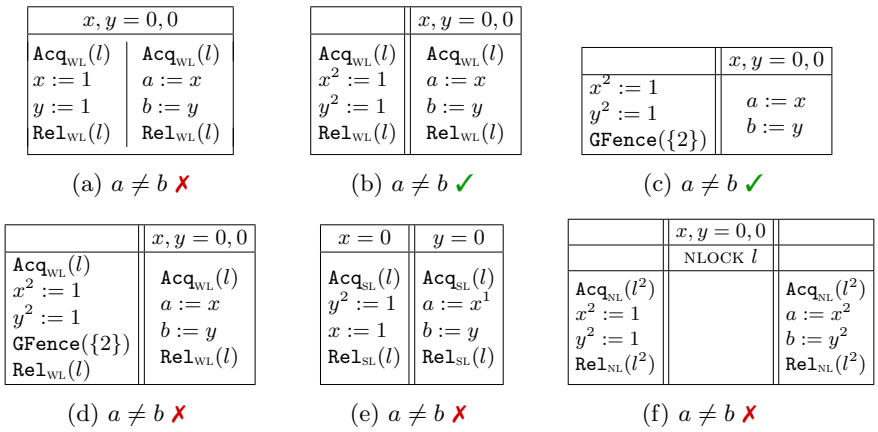


Fig. 8: Examples of weak, strong, and node lock behaviours

issues a global fence towards *every* node on the network to ensure that there are no pending remote operations. This is in contrast to a weak lock release implementation that issues no fence by default, and developers have full control over fencing the relevant nodes.

Node Lock Library. As a midway between the efficient weak locks, requiring manual synchronisation, and the inefficient strong locks, we develop the concept of (more fine-grained) *node* locks. Intuitively, as shown in Fig. 8f, a node lock l is associated with a specific node n (node 2 in Fig. 8f) and provides strong guarantees only for locations on n.

A node lock is stronger than a weak lock: as shown in Fig. 8f the weak behaviour $a \neq b$ is prohibited without the need for additional synchronisation. Moreover, a node lock is weaker than a strong lock in two ways. First, it only provides guarantees for operations towards *one* node. For instance, consider a variant of the example in Fig. 8f where the lock l is associated with node 1 (instead of 2); the outcome $a \neq b$ would once again be allowed. Second, it does *not* provide any intra-thread ordering guarantees in that releasing a node lock does not guarantee that previous operations (even towards the associated node) have completed. For instance, in the program $\texttt{Acq}_{\text{NL}}(l^2); z^2 := x; \texttt{Rel}_{\text{NL}}(l^2); x := 1$ the outcome $z = 1$ is *allowed*: $z^2 := x$ and the lock release may not have fully completed before the CPU runs the subsequent $x := 1$ store; i.e., while $z^2 := x$ cannot be reordered past $\texttt{Rel}_{\text{NL}}(l^2)$, the $x := 1$ can be reordered *before* both of them. More concretely, our implementation of $\texttt{Rel}_{\text{NL}}$ (§4.4) comprises RDMA operations that can be delayed after later CPU operations.

Nevertheless, as a common usage of a lock is to protect a specific object that is likely to reside on a single node, this level of guarantee is sufficient for many applications, while enabling efficient implementations.

The $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ Library. Lastly, to simplify RDMA programming, we specify and implement the $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ library that fully abstracts away the notion of nodes and provides strong *sequentially consistent* (SC) [23] semantics via four (per-location) instructions, $\texttt{Write}_{\text{SC}}$, $\texttt{Read}_{\text{SC}}$, $\texttt{CAS}_{\text{SC}}$, and $\texttt{FAA}_{\text{SC}}$ analogous to stores,

loads, and RMWs on CPUs with strong SC semantics. Our implementation uses node locks to wrap RDMA operations and ensure they become visible in the order they are submitted. Indeed, as we discuss later in §5, we can use the same approach to implement *any concurrent data structure* over RDMA, and show that it is correct in that it is *linearisable* [19].

3 Extending RDMA$^{\text{WAIT}}$ to RDMA$^{\text{WAIT}}_{\text{RMW}}$

We present RDMA$^{\text{WAIT}}_{\text{RMW}}$ *model*, an extension of RDMA$^{\text{WAIT}}$ [4] with remote RMW instructions. Our definitions naturally extend those of RDMA$^{\text{WAIT}}$. To underline the distinction between the two, we have highlighted our extensions from RDMA$^{\text{WAIT}}$ to RDMA$^{\text{WAIT}}_{\text{RMW}}$. We specify RDMA$^{\text{WAIT}}_{\text{RMW}}$ in MOWGLI [4], yielding a *modular* semantics that enables *compositional* reasoning. In particular, as we show below, since LOCO libraries can be freely composed together, this allows us to use the *locality* result of MOWGLI to verify each library modularly (in isolation). We proceed with an account of MOWGLI preliminaries (§3.1) and present RDMA$^{\text{WAIT}}_{\text{RMW}}$ in §3.2.

3.1 The MOWGLI Framework Preliminaries

We assume two sets for threads $t \in \mathsf{Tid}$ and nodes $n \in \mathsf{Node}$, where each thread t is associated with a node $\mathtt{n}(t)$. Recall from §2.2 that MOWGLI can be instantiated with a set of *stamps* Stamp and a relation $\mathsf{sto} \subseteq \mathsf{Stamp} \times \mathsf{Stamp}$. In the case of RDMA$^{\text{WAIT}}$ and RDMA$^{\text{WAIT}}_{\text{RMW}}$, the stamps and their associated sto are as presented in Fig. 9. Note that certain stamps, e.g. $\mathtt{aNLR}_n$, are associated with a node n, and each induce a *family* of stamps, e.g. $\mathtt{aNLR} \triangleq \bigcup_{n \in \mathsf{Node}} \{\mathtt{aNLR}_n\}$. The highlighted sections (row H and column 8) denote our extensions from RDMA$^{\text{WAIT}}$ to RDMA$^{\text{WAIT}}_{\text{RMW}}$ and are associated with the new stamp family `aNAR` used to specify RMWs (see §2.2). The ✓ (e.g. in cell A2) denotes that the corresponding stamps (e.g. `aCR` and `aCW`) are *ordered* in that the program order between their constituent subevents is *preserved* and thus their effects are observed in order. Conversely, ✗ denotes that the stamps are *not ordered* (they may be reordered) and thus the effects of their subevents may be observed *out of order*. The SN denotes the stamps are ordered if and only if they are associated with the *same node*.

Libraries. Intuitively, a library L specification identifies its associated *methods* as well as the semantics of these methods. A *method call* is of the form $m(\widetilde{v})$, where m denotes the method name and $\widetilde{v}$ denote its arguments. Ambal et al. capture the method semantics in MOWGLI by identifying the set of *executions* that are L-*consistent* in that they uphold the guarantees promised by L. To this end, they associate L with a set $\mathcal{C}$ of L-*consistent executions*. A *library* is then formally defined as a triple $L = \langle M, \mathtt{loc}, \mathcal{C} \rangle$, where M is its set of *method names* (e.g. `Write` or `Put`); `loc` associates each method call with its set of accessed locations (within the method call arguments); and $\mathcal{C}$ is its set of L-*consistent executions*. (MOWGLI further requires $\mathcal{C}$ to adhere to some basic properties to ensure modularity [4], which we elide here.) We use the prefix 'L.' to project the components of a library L, e.g. $L.M$.

			Later (in Program Order) Stamp											
		sto	single					families						
			1	2	3	4	5	6	7	8	9	10	11	12
			aCR	aCW	aCAS	aMF	aWT	$\texttt{aNLR}_n$	$\texttt{aNRW}_n$	$\texttt{aNAR}_n$	$\texttt{aNRR}_n$	$\texttt{aNLW}_n$	$\texttt{aRF}_n$	$\texttt{aGF}_n$
Earlier (in Program Order) Stamp: single	A	aCR	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	B	aCW	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
	C	aCAS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	D	aMF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	E	aWT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
families	F	$\texttt{aNLR}_n$	✗	✗	✗	✗	✗	SN	SN	SN	SN	SN	SN	SN
	G	$\texttt{aNRW}_n$	✗	✗	✗	✗	✗	✗	SN	SN	SN	SN	✗	SN
	H	$\texttt{aNAR}_n$	✗	✗	✗	✗	✗	✗	SN	SN	SN	SN	SN	SN
	I	$\texttt{aNRR}_n$	✗	✗	✗	✗	✗	✗	✗	✗	✗	SN	SN	SN
	J	$\texttt{aNLW}_n$	✗	✗	✗	✗	✗	✗	✗	✗	✗	SN	✗	SN
	K	$\texttt{aRF}_n$	✗	✗	✗	✗	✗	SN	SN	SN	SN	SN	SN	SN
	L	$\texttt{aGF}_n$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Fig. 9: The sto order in $\text{RDMA}^{\text{WAIT}}$ and $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$, where highlighted cells denote our extensions from $\text{RDMA}^{\text{WAIT}}$ to $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$. The ✓ denotes that the (program-order-related) stamps are *ordered*; the ✗ denotes that the stamps are *not ordered*; the SN denotes the stamps are ordered iff they are associated with the *same node*.

Events and Executions. In the literature of declarative models, traces of a program are represented as a set of *executions*. An execution is a graph comprising: 1) a set of *events* (graph nodes), where each event is associated with the execution of a method call; and 2) a number of relations on events (graph edges). For instance, if thread t executes a $\texttt{Read}(x)$ and reads value v, the corresponding event is of the form $\langle t, \iota, \langle \texttt{Read}, (x), v\rangle\rangle$, where ι denotes its (unique) event identifier. Identifiers serve to distinguish calls to the same method (with same arguments and output) by the same thread in an execution. For an event e, we write $\mathtt{t}(\mathsf{e})$ and $\mathtt{m}(\mathsf{e})$ to extract its thread and method name, respectively.

Definition 1 (Events and Executions). *An* event *is a tuple* $\langle t, \iota, \langle m, \widetilde{v}, v'\rangle\rangle$, *where* $t \in \mathsf{Tid}$ *denotes the executing thread,* ι *is the (unique) event identifier,* m *denotes the method being executed,* $\widetilde{v} \in \mathsf{Val}^*$ *is the method* input *(arguments) and* $v' \in \mathsf{Val}$ *is its* output *(return value, which may be unit* $()$*). An* execution $\mathcal{G}$ *is a tuple* $\langle E, \mathsf{po}, \mathtt{stmp}, \mathsf{so}, \mathsf{hb}\rangle$ *where:*

- E *is the set of events;*
- $\mathsf{po} \subseteq E \times E$ *is the (strict) program order, total for each thread;*
- $\mathtt{stmp} : E \to \mathcal{P}(\mathsf{Stamp})$ *associates each event with a* non-empty *set of stamps and induces a set of* subevents, $\mathsf{SEvent} \triangleq \{\langle \mathsf{e}, a\rangle \mid \mathsf{e} \in E \wedge a \in \mathtt{stmp}(\mathsf{e})\}$*;*
- $\mathsf{so} \subseteq \mathsf{SEvent} \times \mathsf{SEvent}$ *is the* synchronisation order*, representing the intra-library dependencies exported by each library;*
- $\mathsf{hb} \subseteq \mathsf{SEvent} \times \mathsf{SEvent}$ *is the* happens-before order*, a strict partial order such that* $\mathsf{so} \cup \mathsf{ppo} \subseteq \mathsf{hb}$*, where* $\mathsf{ppo} \subseteq \mathsf{SEvent} \times \mathsf{SEvent}$ *denotes the* preserved program order *capturing inter-library dependencies and is defined as follows:*

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

Notations. Given a set A and a relation $\mathsf{r} \subseteq A \times A$, we write r^+ for the transitive closure of r; r^* for its reflexive transitive closure; r^{-1} for the inverse of r; and $[A]$ for the identity relation on A, i.e. $\{\langle a, a\rangle \mid a \in A\}$. We write $\mathsf{r}_1;\mathsf{r}_2$ for the relational composition of r_1 and r_2: $\{\langle a, b\rangle \mid \exists c.\, \langle a, c\rangle \in \mathsf{r}_1 \wedge \langle c, b\rangle \in \mathsf{r}_2\}$. We write $A|_c$ to restrict A with condition c. For instance, given a set of events E, we define $E|_L \triangleq \{\mathsf{e} \in E \mid \mathtt{m}(\mathsf{e}) \in L.M\}$, $E|_t \triangleq \{\mathsf{e} \in E \mid \mathtt{t}(\mathsf{e}) = t\}$, and we write $E|_d$ for the set of events in E with work identifier d. We define $E_x \triangleq \{\mathsf{e} \in E \mid x \in \mathtt{loc}(\mathsf{e})\}$. Similarly, we define $\mathsf{r}|_c \triangleq [A|_c];\mathsf{r};[A|_c]$ (e.g. $\mathsf{po}|_t$) and $\mathsf{r}_x \triangleq [E_x];\mathsf{r};[E_x]$ (e.g. po_x). Given a subset $A' \subseteq A$, we define $\mathsf{r}|_{A'} \triangleq [A'];\mathsf{r};[A']$. When r is a strict partial order, we write $\mathsf{r}|_{\text{imm}}$ for its immediate edges, i.e. $\mathsf{r} \setminus (\mathsf{r};\mathsf{r})$.

Given execution $\mathcal{G} = \langle E, \mathsf{po}, \mathtt{stmp}, \mathsf{so}, \mathsf{hb}\rangle$, we write $\mathcal{G}|_L$ for $\langle E|_L, \mathsf{po}|_L, \mathtt{stmp}|_L, \mathsf{so}|_L, \mathsf{hb}|_L\rangle$, where $\mathtt{stmp}|_L$ denotes the function obtained by restricting the domain of $\mathtt{stmp}$ (i.e. E) to $E|_L$. We use the prefix '$\mathcal{G}$.' to project the components of $\mathcal{G}$ (e.g. $\mathcal{G}.\mathsf{po}$), including its derived ones (e.g. $\mathcal{G}.\mathsf{SEvent}$). Given a stamp a, we write $\mathcal{G}.a$ for $\{\mathsf{s} \in \mathcal{G}.\mathsf{SEvent} \mid \mathsf{s} = \langle \_, a\rangle\}$; analogously for a stamp family, e.g. $\mathcal{G}.\mathtt{aNRR}$. We define the set of *read subevents* as $\mathcal{G}.\mathcal{R} \triangleq \mathcal{G}.\mathtt{aCR} \cup \mathcal{G}.\mathtt{aCAS} \cup \mathcal{G}.\mathtt{aNLR} \cup \mathcal{G}.\mathtt{aNAR} \cup \mathcal{G}.\mathtt{aNRR}$, and *write subevents* as $\mathcal{G}.\mathcal{W} \triangleq \mathcal{G}.\mathtt{aCW} \cup \mathcal{G}.\mathtt{aCAS} \cup \mathcal{G}.\mathtt{aNLW} \cup \mathcal{G}.\mathtt{aNRW}$. Given a set of subevents A, we define $A_x \triangleq \{\mathsf{s} \in A \mid \mathtt{loc}(\mathsf{s}) = \{x\}\}$; e.g. $\mathcal{G}.\mathcal{W}_x$ is the set of write subevents on x. When the choice of $\mathcal{G}$ is clear, we omit '$\mathcal{G}$.', e.g. we simply write $\mathcal{W}$ for $\mathcal{G}.\mathcal{W}$ and $[\mathtt{aCW}]$ for $[\mathcal{G}.\mathtt{aCW}]$.

Consistency. An execution is *consistent* against a set of libraries Λ iff 1) $\mathcal{G}|_L$ is L-consistent for each $L \in \Lambda$; 2) its events and their synchronisation are those of the libraries in Λ; and 3) its happens-before relation is irreflexive. Note that the first condition ensures *modularity* as each library can specify independently the visible behaviours of its functions (stamps), its allowed outcomes (consistency) and the synchronisation (guarantees) it offers (so).

Definition 2 (Consistency). *Let Λ be a set of libraries where $L_1.M \cap L_2.M = \emptyset$ for distinct L_1, L_2. An execution $\mathcal{G} = \langle E, \mathsf{po}, \mathtt{stmp}, \mathsf{so}, \mathsf{hb}\rangle$ is* Λ-consistent *iff:*

- *For all $L \in \Lambda$: $\mathcal{G}|_L \in L.\mathcal{C}$ (i.e. $\mathcal{G}$ is L-consistent for each $L \in \Lambda$);*
- *$E = \bigcup_{L \in \Lambda} E|_L$ and $\mathsf{so} = \bigcup_{L \in \Lambda} \mathsf{so}|_L$; and*
- *hb is irreflexive (i.e. hb is a strict partial order).*

3.2 The Declarative $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ Model

We present $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ as an extension of $\text{RDMA}^{\text{WAIT}}$ [4] with remote RMWs. Our definitions naturally extend those of $\text{RDMA}^{\text{WAIT}}$. To underline the distinction between the two, we have highlighted our extensions from $\text{RDMA}^{\text{WAIT}}$ to $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$.

The $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ Methods. $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ methods extend those of $\text{RDMA}^{\text{WAIT}}$ with remote RMWs as defined by the following grammar, where $\text{RDMA}^{\text{WAIT}}$ methods comprise local (CPU) operations on TSO machines and remote operations.

$$
\begin{aligned}
m(\widetilde{v}) ::= {} & \mathtt{Write}(x, v) \mid \mathtt{Read}(x) \mid \mathtt{CAS}(x, v_1, v_2) \mid \mathtt{Mfence}() && \text{// } \text{RDMA}^{\text{WAIT}}\text{: Local} \\
\mid {} & \mathtt{Get}(x, y, d) \mid \mathtt{Put}(x, y, d) \mid \mathtt{Wait}(d) \mid \mathtt{Rfence}(n) && \text{// } \text{RDMA}^{\text{WAIT}}\text{: Remote} \\
\mid {} & \mathtt{RCAS}(x, y, v_1, v_2, d) \mid \mathtt{RFAA}(x, y, v, d) && \text{// Remote RMWs}
\end{aligned}
$$

The remote operations comprise `Get`, `Put`, `Wait` (as described in §2.1), and `Rfence` instructions. Note that for readability in our examples we write $x :=^d y^n$ (resp. $x^n :=^d y$) for $\texttt{Get}(x, y, d)$ (resp. $\texttt{Put}(x, y, d)$). Similarly, we write $x := v$ (resp. $a := x$) for $\texttt{Write}(x, v)$ (resp. $\texttt{let}\, a = \texttt{Read}(x)\, \texttt{in} \ldots$). The $\texttt{Rfence}(n)$ denotes a *remote fence* that strongly orders all operations towards n without blocking the (local) CPU. That is, given a (sequential) program of the form $C; \texttt{Rfence}(n); C'$, all remote operations towards n in C are ordered before those in C'. The $\texttt{RCAS}(x, y, v_1, v_2, d)$ is the remote analogue of writing $\texttt{let}\, v = \texttt{CAS}(y, v_1, v_2)\, \texttt{in}\, \texttt{Write}(x, v)$ with work identifier d, where the RMW is run on remote location y and the result is written to local location x. Similarly, $\texttt{RFAA}(x, y, v, d)$ increments (remote) y by v and writes its old value to x.

Well-addressed $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ **Executions.** We assume each location x is associated with exactly one node denoted by $\texttt{n}(x)$. We write $\texttt{n}(t)$ to denote the node on which t is run. An execution $\mathcal{G}$ is *well-addressed* iff it comprises method calls (in $\mathcal{G}.E$) with appropriate local locations when expected; e.g. for each $\texttt{Write}(x, \_)$ or $\texttt{Put}(\_, x, \_)$ call by thread t in $\mathcal{G}$, $\texttt{n}(x)$=$\texttt{n}(t)$. We define `loc` for $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ as expected; e.g. $\texttt{loc}(\texttt{Write}(x, \_))$=$\{x\}$, $\texttt{loc}(\texttt{Put}(x, y, \_))$=$\{x, y\}$ and $\texttt{loc}(\texttt{Mfence})$=$\emptyset$.

Well-stamped $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ **Executions.** An execution $\mathcal{G}$ is *well-stamped* if for all $\texttt{e} = \langle \_, \_, \langle m, (\widetilde{v}), v' \rangle \rangle \in \mathcal{G}.E$: $\mathcal{G}.\texttt{stmp}(\texttt{e}) \in \texttt{stmp}_{\texttt{RW}}(m(\widetilde{v}), v')$, with $\texttt{stmp}_{\texttt{RW}}$ defined as follows. Note that depending on whether `RCAS` calls succeed, they may have multiple valid sets of stamps; as such, the $\texttt{stmp}_{\texttt{RW}}$ function returns a set of stamp sets (set of set of stamps), though in all cases but for `RCAS` this set is a singleton.

$$\texttt{stmp}_{\texttt{RW}}(\texttt{CAS}(x, v_1, \_), v_2) \triangleq \begin{cases} \{\{\texttt{aMF}, \texttt{aCR}\}\} & \text{if } v_1 \neq v_2 \\ \{\{\texttt{aCAS}\}\} & \text{if } v_1 = v_2 \end{cases} \qquad \texttt{stmp}_{\texttt{RW}}(\texttt{Write}(x, v), \_) \triangleq \{\{\texttt{aCW}\}\}$$

$$\texttt{stmp}_{\texttt{RW}}(\texttt{Read}(x), \_) \triangleq \{\{\texttt{aCR}\}\} \qquad \texttt{stmp}_{\texttt{RW}}(\texttt{Get}(x, y, \_), \_) \triangleq \{\{\texttt{aNRR}_{\texttt{n}(y)}, \texttt{aNLW}_{\texttt{n}(y)}\}\}$$

$$\texttt{stmp}_{\texttt{RW}}(\texttt{Mfence}(), \_) \triangleq \{\{\texttt{aMF}\}\} \qquad \texttt{stmp}_{\texttt{RW}}(\texttt{Put}(x, y, \_), \_) \triangleq \{\{\texttt{aNLR}_{\texttt{n}(x)}, \texttt{aNRW}_{\texttt{n}(x)}\}\}$$

$$\texttt{stmp}_{\texttt{RW}}(\texttt{Wait}(d), \_) \triangleq \{\{\texttt{aWT}\}\} \qquad \texttt{stmp}_{\texttt{RW}}(\texttt{RFAA}(x, y^n, \_, \_), \_) \triangleq \{\{\texttt{aNAR}_n, \texttt{aNRW}_n, \texttt{aNLW}_n\}\}$$

$$\texttt{stmp}_{\texttt{RW}}(\texttt{Rfence}(n), \_) \triangleq \{\{\texttt{aRF}_n\}\} \qquad \texttt{stmp}_{\texttt{RW}}(\texttt{RCAS}(x, y, \_, \_, \_), \_) \triangleq \{\{\texttt{aNAR}_{\texttt{n}(y)}, \texttt{aNLW}_{\texttt{n}(y)}\}, \{\texttt{aNAR}_{\texttt{n}(y)}, \texttt{aNRW}_{\texttt{n}(y)}, \texttt{aNLW}_{\texttt{n}(y)}\}\}$$

A successful remote RMW has three stamps for reading the remote location, modifying it, and writing it to the local location, while a failed `RCAS` does not modify the remote location. Recall that the remote read of a remote RMW yields stamp $\texttt{aNAR}_n$, which offers more guarantees than the stamp $\texttt{aNRR}_n$ of `Get`s.

We extend the location function (`loc`, defined above for $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$) to subevents. For method calls corresponding to *local operations* (with one or zero locations) their subevents have the same locations. The subevents of `Get`, `Put`, `RCAS`, and `RFAA` are associated with the relevant location as expected. For instance, if $\texttt{e} = \langle \_, \_, \langle \texttt{Get}, (x, y, d), \_ \rangle \rangle$ (with subevents $\texttt{aNRR}_n$ and $\texttt{aNLW}_n$), then $\texttt{loc}(\langle \texttt{e}, \texttt{aNRR}_n \rangle) = \{y\}$ and $\texttt{loc}(\langle \texttt{e}, \texttt{aNLW}_n \rangle) = \{x\}$; whereas if $\texttt{e} = \langle \_, \_, \langle \texttt{RFAA}, (x, y, v, d), \_ \rangle \rangle$, then $\texttt{loc}(\langle \texttt{e}, \texttt{aNAR}_n \rangle)$=$\{y\}$, $\texttt{loc}(\langle \texttt{e}, \texttt{aNRW}_n \rangle)$=$\{y\}$ and $\texttt{loc}(\langle \texttt{e}, \texttt{aNLW}_n \rangle)$=$\{x\}$.

Well-formed $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ **Executions.** We shortly define the notion of $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$-consistency for an execution $\mathcal{G}$. To do this, we need a few auxiliary functions and relations as follows. We assume functions $\texttt{v}_\texttt{R} : \mathcal{G}.\mathcal{R} \to \mathsf{Val}$ and $\texttt{v}_\texttt{W} : \mathcal{G}.\mathcal{W} \to \mathsf{Val}$, which associate each read (resp. write) subevent with the value returned (resp. written). We define the '*reads-from*' relation, $\mathsf{rf} \subseteq \mathcal{G}.\mathcal{W} \times \mathcal{G}.\mathcal{R}$, on subevents of

the same location with matching values (formalised below); the '*modification-order*' relation, $\mathsf{mo} \subseteq \mathcal{G}.\mathcal{W} \times \mathcal{G}.\mathcal{W}$, describing a (total) order in which writes reach the memory; and the '*NIC flush order*', nfo, capturing the PCIe guarantee that NIC reads flush previous NIC writes. For remote RMWs, we define the '*remote-atomic-order*', rao, describing the (total) order in which remote read parts of remote RMWs towards each node is executed. A tuple $\langle \mathtt{v_R}, \mathtt{v_W}, \mathsf{rf}, \mathsf{mo}, \mathsf{nfo}, \mathsf{rao} \rangle$ is *well-formed* if the following hold for all $\mathtt{e}, v, v', v_1, v_2, \mathtt{s}_1, \mathtt{s}_2, n, x, y$.

- If $\mathtt{e}$ is of the form $\langle \mathtt{Read}, \_, v \rangle$ or $\langle \mathtt{CAS}, \_, v \rangle$, then $\mathtt{v_R}(\mathtt{e}) = v$.
- If $\mathtt{e}$ is of the form $\langle \mathtt{Write}, (\_, v), \_ \rangle$ or $\langle \mathtt{CAS}, (\_, v', v), v' \rangle$, then $\mathtt{v_W}(\mathtt{e}) = v$
- If $\mathtt{s}_1 = \langle \mathtt{e}, \mathtt{aNLR}_n \rangle \wedge \mathtt{s}_2 = \langle \mathtt{e}, \mathtt{aNRW}_n \rangle$, then $\mathtt{v_R}(\mathtt{s}_1) = \mathtt{v_W}(\mathtt{s}_2)$; *mutatis mutandis* for $\mathtt{s}_1 = \langle \mathtt{e}, \mathtt{aNRR}_n \rangle, \mathtt{s}_2 = \langle \mathtt{e}, \mathtt{aNLW}_n \rangle$ and $\mathtt{s}_1 = \langle \mathtt{e}, \mathtt{aNAR}_n \rangle, \mathtt{s}_2 = \langle \mathtt{e}, \mathtt{aNLW}_n \rangle$.
- $\langle \mathtt{s}_1, \mathtt{s}_2 \rangle \in \mathsf{rf} \Rightarrow \mathtt{loc}(\mathtt{s}_1) = \mathtt{loc}(\mathtt{s}_2) \wedge \mathtt{v_W}(\mathtt{s}_1) = \mathtt{v_R}(\mathtt{s}_2)$.
- rf^{-1} is a function, i.e. every read is related to at most one write. If a read is not related to a write, it returns zero: $\mathtt{s}_2 \notin \mathtt{img}(\mathsf{rf}) \Rightarrow \mathtt{v_R}(\mathtt{s}_2) = 0$.
- $\mathsf{mo} \triangleq \bigcup_{x \in \mathsf{Loc}} \mathsf{mo}_x$, where each mo_x is a strict total order on $\mathcal{G}.\mathcal{W}_x$.
- if $\langle \mathtt{s}_1, \mathtt{s}_2 \rangle \in (\mathtt{aNLR}_n \times \mathtt{aNLW}_n) \cup ((\mathtt{aNRR}_n \cup \mathtt{aNAR}_n) \times \mathtt{aNRW}_n)$ and $\mathtt{t}(\mathtt{s}_1) = \mathtt{t}(\mathtt{s}_2)$ then $\langle \mathtt{s}_1, \mathtt{s}_2 \rangle \in \mathsf{nfo} \cup \mathsf{nfo}^{-1}$.
- $\mathtt{RCAS}$ succeeds iff it reads the expected value, in which case it overwrites with the given value. That is, given $\mathtt{e} = \langle \_, \_, \langle \mathtt{RCAS}, (x, y, v_1, v_2, \_), \_ \rangle \rangle$:
  if $\mathtt{stmp}(\mathtt{e}) = \{\mathtt{aNAR}_{\mathtt{n}(y)}, \mathtt{aNLW}_{\mathtt{n}(y)}\}$, then $\mathtt{v_R}(\langle \mathtt{e}, \mathtt{aNAR}_{\mathtt{n}(y)} \rangle) \neq v_1$; and
  if $\mathtt{stmp}(\mathtt{e}) = \{\mathtt{aNAR}_{\mathtt{n}(y)}, \mathtt{aNRW}_{\mathtt{n}(y)}, \mathtt{aNLW}_{\mathtt{n}(y)}\}$, then $\mathtt{v_R}(\langle \mathtt{e}, \mathtt{aNAR}_{\mathtt{n}(y)} \rangle) = v_1$ and $\mathtt{v_W}(\langle \mathtt{e}, \mathtt{aNRW}_{\mathtt{n}(y)} \rangle) = v_2$.
- If $\mathtt{e} = \langle \_, \_, \langle \mathtt{RFAA}, (x, y, v, \_), \_ \rangle \rangle$, then $\mathtt{v_W}(\langle \mathtt{e}, \mathtt{aNRW}_{\mathtt{n}(y)} \rangle) = \mathtt{v_R}(\langle \mathtt{e}, \mathtt{aNAR}_{\mathtt{n}(y)} \rangle) + v$.
- $\mathsf{rao} \triangleq \bigcup_{n \in \mathsf{Node}} \mathsf{rao}_n$, where rao_n is a strict total order on the set of subevents $\{\langle \mathtt{e}, \mathtt{aNAR}_n \rangle \mid \mathtt{e} = \langle \_, \_, \langle m, (x, y, \ldots), \_ \rangle \rangle \wedge m \in \{\mathtt{RFAA}, \mathtt{RCAS}\} \wedge \mathtt{n}(y) = n\}$

We distinguish the point subevents *start* executing (point of 'issue') from when they *complete*. We define the *issued-before* relation, ib, to record dependencies between the starts of subevents, while so records dependencies between their ends. Note that ib and so are incomparable: $\langle \mathtt{s}_1, \mathtt{s}_2 \rangle \in \mathsf{ib}$ does *not* imply $\langle \mathtt{s}_1, \mathtt{s}_2 \rangle \in \mathsf{so}$ and vice versa. We define *instantaneous subevents*, $\mathcal{G}.\mathtt{Inst} \triangleq \mathcal{G}.\mathsf{SEvent} \setminus (\mathcal{G}.\mathtt{aCW} \cup \mathcal{G}.\mathtt{aNLW} \cup \mathcal{G}.\mathtt{aNRW})$, as those that start and end at the same time.

Given an execution $\mathcal{G}$ and well-formed $\langle \mathtt{v_R}, \mathtt{v_W}, \mathsf{rf}, \mathsf{mo}, \mathsf{nfo}, \mathsf{rao} \rangle$, we further define the following relations that will help us define ib and so for $\mathrm{RDMA}^{\mathrm{WAIT}}_{\mathrm{RMW}}$:

$$\mathsf{rb} \triangleq \left\{ \langle r, w \rangle \in \mathcal{G}.\mathcal{R} \times \mathcal{G}.\mathcal{W} \;\middle|\; \begin{array}{l} (\langle r, w \rangle \in (\mathsf{rf}^{-1}; \mathsf{mo}) \vee r \notin \mathtt{img}(\mathsf{rf})) \\ \wedge\ \mathtt{loc}(r) = \mathtt{loc}(w) \end{array} \right\} \setminus [\mathcal{G}.\mathsf{SEvent}]$$

$$\mathsf{rb}_{\mathsf{i}} \triangleq [\mathtt{aCR}]; ((\mathsf{po} \cup \mathsf{po}^{-1}) \cap \mathsf{rb}); [\mathtt{aCW}] \qquad \mathsf{pfg} \triangleq \{\langle \langle \mathtt{e}_1, \mathtt{aNLW}_n \rangle, \langle \mathtt{e}_2, \mathtt{aWT} \rangle \rangle \mid \exists d. \langle \mathtt{e}_1, \mathtt{e}_2 \rangle \in \mathsf{po}|_d\}$$

$$\mathsf{rf}_{\mathsf{i}} \triangleq [\mathtt{aCW}]; (\mathsf{po} \cap \mathsf{rf}); [\mathtt{aCR}] \qquad \mathsf{rf}_{\mathsf{e}} \triangleq \mathsf{rf} \setminus \mathsf{rf}_{\mathsf{i}} \qquad \mathsf{pfp} \triangleq \{\langle \langle \mathtt{e}_1, \mathtt{aNRW}_n \rangle, \langle \mathtt{e}_2, \mathtt{aWT} \rangle \rangle \mid \exists d. \langle \mathtt{e}_1, \mathtt{e}_2 \rangle \in \mathsf{po}|_d\}$$

$$\begin{aligned} \mathsf{iso} \triangleq\ & \{\langle \langle \mathtt{e}, \mathtt{aMF} \rangle, \langle \mathtt{e}, \mathtt{aCR} \rangle \rangle \mid \mathtt{m}(\mathtt{e}) = \mathtt{CAS}\} \\ & \cup \{\langle \langle \mathtt{e}, \mathtt{aNRR}_n \rangle, \langle \mathtt{e}, \mathtt{aNLW}_n \rangle \rangle \mid \mathtt{m}(\mathtt{e}) = \mathtt{Get}\} \cup \{\langle \langle \mathtt{e}, \mathtt{aNLR}_n \rangle, \langle \mathtt{e}, \mathtt{aNRW}_n \rangle \rangle \mid \mathtt{m}(\mathtt{e}) = \mathtt{Put}\} \\ & \cup \{\langle \langle \mathtt{e}, \mathtt{aNAR}_n \rangle, \langle \mathtt{e}, \mathtt{aNLW}_n \rangle \rangle \mid \mathtt{m}(\mathtt{e}) \in \{\mathtt{RCAS}, \mathtt{RFAA}\}\} \\ & \cup \{\langle \langle \mathtt{e}, \mathtt{aNAR}_n \rangle, \langle \mathtt{e}, \mathtt{aNRW}_n \rangle \rangle \mid \mathtt{m}(\mathtt{e}) \in \{\mathtt{RCAS}, \mathtt{RFAA}\} \wedge \mathtt{aNRW}_n \in \mathtt{stmp}(\mathtt{e})\} \end{aligned}$$

The rb denotes the '*reads-before*' relation: given a read r that reads from a write w_r, i.e. $\langle w_r, r \rangle \in \mathsf{rf}$, then rb relates r to all writes w (on the same location) that are mo-*later* than w_r. The *internal* rb relation, rb_{i}, restricts rb to CPU reads

and writes on the same thread; similarly for $\mathsf{rf_i}$ (internal rf). The *external* rf, $\mathsf{rf_e}$, is defined as rf edges that are not internal. The pfg (resp. pfp) relation captures the synchronisation between the local write subevent of a `Get` or remote RMW (resp. remote write subevent of a `Put` or remote RMW) and a later `Wait` with the same work identifier. As we describe shortly, while both are included in ib, only pfg is included in so as waiting for a `Put` (or remote RMW) does not guarantee that the NIC remote write has completed. The '*internal synchronisation order*', iso, captures ordering between subevents of the same event and ensure that a failing CPU `CAS` performs a memory fence before reading; RDMA operations (`Get`, `Put`, and remote RMW) read before copying the value; and a successful remote RMW reads before updating the remote value.

Finally, we define ib as follows and it includes a superset ippo of ppo. Specifically, while a later CPU read might finish before an earlier CPU write or wait (cells B1 and B5, in Fig. 9), they start (are issued) in order; and while a remote fence does not guarantee previous NIC writes have completed (cells G11 and J11, in Fig. 9), it guarantees they have at least started.

$$\mathsf{ib} \triangleq (\mathsf{ippo} \cup \mathsf{iso} \cup \mathsf{rf} \cup \mathsf{pfg} \cup \mathsf{pfp} \cup \mathsf{nfo} \cup \mathsf{rb_i})^+$$
$$\text{with } \mathsf{ippo} \triangleq \mathsf{ppo} \cup ([\mathtt{aCW}]; \mathsf{po}; [\mathtt{aCR} \cup \mathtt{aWT}]) \cup \bigcup_{n \in \mathsf{Node}} ([\mathtt{aNRW}_n \cup \mathtt{aNLW}_n]; \mathsf{po}; [\mathtt{aRF}_n])$$

We next define *consistency* for $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$. We require that ib and so be irreflexive (the latter is implied by irreflexivity of hb in Def. 2 as $\mathsf{so} \subseteq \mathsf{hb}$ (Def. 1)).

Definition 3 ($\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$**-consistency).** *An execution* $\mathcal{G} = \langle E, \mathsf{po}, \mathtt{stmp}, \mathsf{so}, \mathsf{hb} \rangle$ *is* $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$-consistent *iff it is well-addressed, well-stamped, and there exists a well-formed tuple* $\langle \mathtt{v_R}, \mathtt{v_W}, \mathsf{rf}, \mathsf{mo}, \mathsf{nfo}, \mathsf{rao} \rangle$ *such that:*

1) ib *is irreflexive; and*
2) $\mathsf{so} = \mathsf{iso} \cup \mathsf{rf_e} \cup \mathsf{pfg} \cup \mathsf{nfo} \cup \mathsf{rb} \cup \mathsf{mo} \cup \mathsf{rao} \cup ([\mathtt{aNRW}]; \mathsf{iso}^{-1}; \mathsf{rao}) \cup ([\mathtt{Inst}]; \mathsf{ib})$.

As described above, rao captures the order in which remote read parts of remote RMWs towards a node is executed. The extension $([\mathtt{aNRW}]; \mathsf{iso}^{-1}; \mathsf{rao})$ ensures that remote RMWs towards the same node do not overlap: if a remote RMW succeeds, then its remote write completes before the next RMW can read.

4 Specifying and Verifying RDMA Lock Libraries

We use the $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ library to *specify, implement, and verify* three RDMA lock libraries. As discussed in §2.4, designing an RDMA lock presents a trade-off between strong, intuitive behaviours and efficient implementations. As such, after introducing the required preliminaries (§4.1), we develop a weak (WLOCK), strong (SLOCK), and node (NLOCK) lock library.

4.1 Preliminaries

Well-formed Locks. A lock library typically provides two methods $\mathtt{Acq}(x)$ and $\mathtt{Rel}(x)$ for acquiring and releasing a (network-shared) lock x, ensuring mutual

exclusion; i.e. no two thread can hold the lock on x simultaneously. We assume the existence of a *location function* $\texttt{loc}$ such that $\texttt{loc}(\texttt{Acq}(x)) = \texttt{loc}(\texttt{Rel}(x)) = \{x\}$. We further assume that locks are used in a *well-formed* fashion: a thread only acquires (resp. releases) lock x if it has not (resp. has) already acquired x. We formalise this in Def. 4 below, requiring that each $\texttt{Acq}(x)$ (resp. $\texttt{Rel}(x)$) is followed (resp. preceded) by $\texttt{Rel}(x)$ (resp. $\texttt{Acq}(x)$) in program order.

Definition 4. *An execution* $\langle E, \mathsf{po}, \_, \_, \_\rangle$ *is* lock-well-formed *iff for all* x*:*

1) for all $\mathtt{e}_a \in E_x$ *there exists an* $\mathtt{e}_r \in E_x$ *such that* $\langle \mathtt{e}_a, \mathtt{e}_r\rangle \in \mathsf{po}_x|_{imm}$*; and*
2) for all $\mathtt{e}_r \in E_x$ *there exists an* $\mathtt{e}_a \in E_x$ *such that* $\langle \mathtt{e}_a, \mathtt{e}_r\rangle \in \mathsf{po}_x|_{imm}$

where $\mathtt{e}_a, \mathtt{e}_r$ *are acquire and release events:* $\mathtt{m}(\mathtt{e}_a) = \texttt{Acq}$ *and* $\mathtt{m}(\mathtt{e}_r) = \texttt{Rel}$.

Library guarantees only hold for programs that adhere to this well-formedness requirement. For those that do not, *any* behaviour is allowed.

Background: SV **Library.** Ambal et al. [4] use RDMA$^{\text{WAIT}}$ to define higher-level libraries such as a *shared-variable* library (SV) where each node maintains its own *copy* for each location x. A thread then accesses (reads/writes) its own local copies, and can broadcast its local value to other nodes. The SV library comprises these methods: $M = \{\texttt{Write}_{\text{SV}}, \texttt{Read}_{\text{SV}}, \texttt{Bcast}_{\text{SV}}, \texttt{Wait}_{\text{SV}}, \texttt{GFence}\}$. The $\texttt{Write}_{\text{SV}}(x, v)$ (resp. $\texttt{Read}_{\text{SV}}(x)$) writes (resp. reads) value v to the local copy of x on the current node. The $\texttt{Bcast}_{\text{SV}}(x, d, \{n_1, \ldots, n_k\})$ broadcasts the local value of x and overwrites x on nodes $n_1, \ldots, n_k$, which may include the local node itself (where d is the work id). The $\texttt{Wait}_{\text{SV}}(d)$ waits for previous broadcasts of the thread associated with work id $d \in \mathsf{Wid}$. Finally, the global fence $\texttt{GFence}(\{n_1, \ldots, n_k\})$ ensures every previous operation of the thread towards nodes $n_1, \ldots, n_k$ is fully completed. We repeat the formal semantics of SV in §A. In the remainder of this article we use SV to implement several libraries.

4.2 The Weak Lock Library

We present our WLOCK library, which only guarantees *mutual exclusion*, without any guarantees on the completion order of submitted RDMA operations.

The WLOCK **Specification.** The stamps for WLOCK are defined through the $\texttt{stmp}_{\text{WL}}$ function as follows. That is, acquiring a weak lock behaves as a memory fence (stamp $\texttt{aMF}$) on TSO, while releasing it behaves merely as a write ($\texttt{aCW}$).

$$\texttt{stmp}_{\text{WL}}(\langle t, \_, \langle \texttt{Acq}_{\text{WL}}, (x), ()\rangle\rangle) \triangleq \{\texttt{aMF}\} \qquad \texttt{stmp}_{\text{WL}}(\langle t, \_, \langle \texttt{Rel}_{\text{WL}}, (x), ()\rangle\rangle) \triangleq \{\texttt{aCW}\}$$

As we formulate in Def. 5 below (the second condition), WLOCK provides synchronisation between lock releases and acquisitions of each lock.

Definition 5 (WLOCK**-consistency).** *A lock-well-formed execution* $\mathcal{G} = \langle E, \mathsf{po}, \texttt{stmp}, \mathsf{so}, \mathsf{hb}\rangle$ *is* WLOCK-consistent *iff:*

1) $\texttt{stmp} = \texttt{stmp}_{\text{WL}}$ *(where* $\texttt{stmp}_{\text{WL}}$ *is as defined above); and*
2) $\mathsf{so} = \bigcup_x \left\{ \langle \langle \mathtt{e}_1, \texttt{aCW}\rangle, \langle \mathtt{e}_2, \texttt{aMF}\rangle\rangle \;\middle|\; \langle \mathtt{e}_1, \mathtt{e}_2\rangle \in (\mathsf{po}_x|_{imm})^{-1}; \mathsf{lo}_x \right\}$*, where* lo_x *is a total order on acquisition events on* x*, i.e. on* $\{\mathtt{e} \in E_x \mid \mathtt{m}(\mathtt{e}) = \texttt{Acq}_{\text{WL}}\}$.

$I_{\texttt{WL}}(t, \texttt{Acq}_{\text{WL}}, (x)) \triangleq$
$\texttt{RFAA}(p_x^t, x_a, 1, d);\ \texttt{Wait}(d);$
$\texttt{let}\ v = \texttt{Read}(p_x^t)\ \texttt{in}$
$\texttt{loop}\ \{\texttt{if}\ \texttt{Read}_{\text{SV}}(x_1) = v\ \texttt{then break else}$
$\quad \ldots$
$\quad \texttt{if}\ \texttt{Read}_{\text{SV}}(x_T) = v\ \texttt{then break}\ \}$

$I_{\texttt{WL}}(t, \texttt{Rel}_{\text{WL}}, (x)) \triangleq$
$\texttt{let}\ v = \texttt{Read}(p_x^t)\ \texttt{in}$
$\texttt{Write}_{\text{SV}}(x_t, v+1);$
$\texttt{Bcast}_{\text{SV}}(x_t, \_, \mathsf{Node} \setminus \{\texttt{n}(t)\})$

Fig. 10: The WLOCK implementation using $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ and SV libraries.

Given a release event e_1 on x (in a lock-well-formed execution), the $(\mathsf{po}_x|_{\text{imm}})^{-1}$ component identifies an acquire event e_3 that is the latest corresponding acquire event on x preceding e_1 (in po). As such, so induces synchronisation between e_1 and all later (in lo_x) acquisition events e_2. Note that lo_x is also indirectly included in hb, since the acquire and release operations stay in order.

The release stamp ($\texttt{aCW}$) does not synchronise with previous RDMA-specific stamps (bottom-left part of Fig. 9). As such, reacquiring a lock does not guarantee that previous RDMA operations submitted with the lock are completed.

The WLOCK (Distributed) Implementation. We present our WLOCK implementation in Fig. 10 (via the $I_{\texttt{WL}}$ function), inspired by the well-known ticket lock implementation. For each lock location x, we create a ticket dispenser x_a (on some arbitrary node) that records the value of the next *available* ticket, thread-local locations (p_x^t for each $t \in \mathsf{Tid} = \{1, \ldots, T\}$) to track the ticket allocated to t (i.e. its turn), and shared variables x_t (for each $t \in \mathsf{Tid}$) to signal releasing the lock.

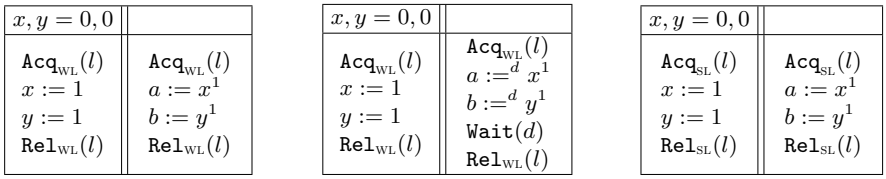
To release the lock on x, thread t writes the *next* turn, i.e. $v{+}1$ when t holds ticket v (obtained by reading p_x^t), to its release location x_t and subsequently broadcasts it to all nodes other than itself ($\texttt{n}(t)$). To acquire the lock on x, thread t calls a fetch-and-add on x_a to fetch the next available ticket (i.e. its turn) in p_x^t and increments x_a. It then records its turn in v and repeatedly examines the release location $x_{t'}$ of each thread $t' \in \{1, \ldots, T\}$ until one has value v, indicating that its turn has come and thus t holds the lock. Note that t' may be t itself, i.e. $t = t'$, if it was the last thread to release the lock.

At the cost of more network messages (through broadcasts), our implementation provides lower latency than centralised systems (e.g. in Fig. 13) as messages are transmitted directly from the thread releasing the lock to the next thread acquiring the lock. We next prove (Theorem 1) that our implementation is correct against the WLOCK specification with the full proof given in §B.2.

Theorem 1. *The implementation $I_{\texttt{WL}}$ is sound.*

4.3 The Strong Lock Library

We present our strong lock library SLOCK that, as well as ensuring mutual exclusion of critical sections, additionally guarantees that *all* earlier operations have *fully* completed on releasing a strong lock. We present several examples of the



(a) $a \neq b$ ✓ (b) $a \neq b$ ✗ (c) $a \neq b$ ✗

Fig. 11: Weak versus strong locks when interacting with `Get` instructions.

'message-passing' behaviour in Fig. 11 contrasting the behaviour of weak and strong locks when interacting with `Get`s and whether the weak outcome $a \neq b$ is allowed. In particular, we may observe $a \neq b$ when using a weak lock (Fig. 11a) and this can be prohibited by explicitly waiting (using $\texttt{Wait}(d)$) on the completion of the `Get`s before releasing the weak lock (Fig. 11b). By contrast, when using a strong lock we no longer need to wait for their completion as this is guaranteed by the strong lock release (Fig. 11c).

The SLOCK Specification. The SLOCK stamps are defined (via $\texttt{stmp}_{\text{SL}}$) as:

$$\texttt{stmp}_{\text{SL}}(\langle t, \_, \langle \texttt{Acq}_{\text{SL}}, (x), () \rangle \rangle) \triangleq \{\texttt{aMF}\} \quad \texttt{stmp}_{\text{SL}}(\langle t, \_, \langle \texttt{Rel}_{\text{SL}}, (x), () \rangle \rangle) \triangleq \bigcup_{n \in \mathsf{Node}} \{\texttt{aGF}_n\}$$

As with WLOCK, acquiring a strong lock behaves as a memory fence (`aMF`), while releasing it behaves as a global fence (`aGF`), ensuring that all previous remote operations are completed.

Definition 6 (SLOCK-consistency). *A lock-well-formed execution* $\mathcal{G} = \langle E, \mathsf{po}, \texttt{stmp}, \mathsf{so}, \mathsf{hb} \rangle$ *is* SLOCK-consistent *iff:*

1) $\texttt{stmp} = \texttt{stmp}_{\text{SL}}$ *(where* $\texttt{stmp}_{\text{SL}}$ *is defined above); and*
2) $\mathsf{so} = \bigcup_{x \in \mathsf{Loc}, n \in \mathsf{Node}} \left\{ \langle \langle \mathsf{e}_1, \texttt{aGF}_n \rangle, \langle \mathsf{e}_2, \texttt{aMF} \rangle \rangle \mid \langle \mathsf{e}_1, \mathsf{e}_2 \rangle \in (\mathsf{po}_x|_{imm})^{-1}; \mathsf{lo}_x \right\}$, *where* lo_x *is a total order on* $\{\mathsf{e} \in E_x \mid \texttt{m}(\mathsf{e}) = \texttt{Acq}_{\text{SL}}\}$.

Strong Lock Implementation. We implement SLOCK (via I_{SL}) simply by combining the weak locks and global fences (from the SV library) as follows:

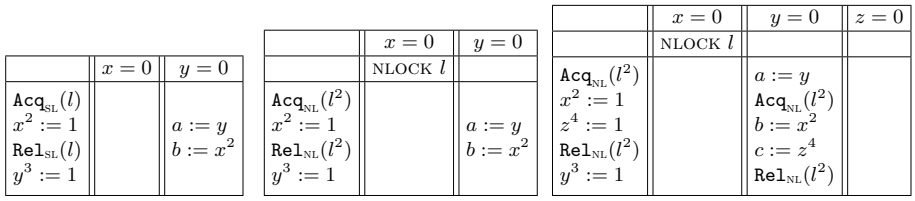
$$I_{\text{SL}}(t, \texttt{Acq}_{\text{SL}}, (x)) \triangleq \texttt{Acq}_{\text{WL}}(x) \qquad I_{\text{SL}}(t, \texttt{Rel}_{\text{SL}}, (x)) \triangleq \texttt{GFence}(\mathsf{Node}); \texttt{Rel}_{\text{WL}}(x)$$

Finally, we prove (Theorem 2) that our implementation is sound against the SLOCK specification with the full proof given in §B.3.

Theorem 2. *The implementation* I_{SL} *is sound.*

4.4 The Node Lock Library

A common use case of locks is to protect an object (set of locations) on a specific node. In such cases, neither weak nor strong locks are suitable as they either incur a high programmer burden (weak locks) or a high performance overhead (strong



(a) $(a, b) = (1, 0)$ ✗ (b) $(a, b) = (1, 0)$ ✓ (c) $(a, b, c) = (1, 0, \_)$ ✗ $(a, b, c) = (1, \_, 0)$ ✓

Fig. 12: Strong (left) versus node (middle and right) locks examples.

locks). To address this, we develop *node locks*, NLOCK, a novel lock library that provides synchronisation on a specific node. Given a node lock x on node n, we write $\mathtt{n}(x)$ for n. A node lock x ensures that on re-acquiring it all previous remote operations (within a critical section of x) towards n are observable.

The NLOCK Specification. The NLOCK stamps are defined (via $\mathtt{stmp}_{\mathtt{NL}}$) as:

$$\mathtt{stmp}_{\mathtt{NL}}(\langle t, \_, \langle \mathtt{Acq}_{\mathtt{NL}}, (x), () \rangle \rangle) \triangleq \{\mathtt{aMF}\} \qquad \mathtt{stmp}_{\mathtt{NL}}(\langle t, \_, \langle \mathtt{Rel}_{\mathtt{NL}}, (x), () \rangle \rangle) \triangleq \{\mathtt{aRF}_{\mathtt{n}(x)}, \mathtt{aNRW}_{\mathtt{n}(x)}\}$$

Note that unlike WLOCK, the NLOCK releases use $\mathtt{aRF}_n$ and $\mathtt{aNRW}_n$ stamps to synchronise with previous remote operations towards n (i.e. those with stamps $\mathtt{aNAR}_n$, $\mathtt{aNRR}_n$, and $\mathtt{aNRW}_n$). Importantly, note that unlike in SLOCK, the release *should not* include a global fence stamp ($\mathtt{aGF}_n$) as that would be too strong. By using $\mathtt{aRF}_n$ and $\mathtt{aNRW}_n$, we ensure that previous operations towards n are completed only when the lock is *later re-acquired*, and they may not have yet completed on release. This means that, when appropriate, using a node lock is more efficient than combining a weak lock with a global fence.

To understand the difference between strong and node locks, consider the examples in Figs. 12a and 12b, where the $x^2 := 1$ `Put` by node 1 is enclosed *within* a lock, while the $b := x^2$ `Get` by node 3 is *without* a lock. In Fig. 12a, $\mathtt{Rel}_{\mathrm{SL}}(l)$ ensures that the earlier $x^2 := 1$ has completed. As such, $a = 1$ implies that $y^3 := 1$ has been executed and that x (in node 2) has been modified, ensuring $b = 1$. By contrast, the $\mathtt{Rel}_{\mathrm{NL}}(l^2)$ in Fig. 12b does not wait for $x^2 := 1$ to complete, i.e. $x^2 := 1$ may complete after $y^3 := 1$. We can prevent this by enclosing $b := x^2$ within the node lock, as shown in Fig. 12c. Specifically, $y^3 := 1$ in Fig. 12c may still complete before earlier remote operations. However, $a = 1$ implies that $y^3 := 1$ is executed, and that thread 1 has at least acquired the lock. As such, when thread 3 acquires the lock via $\mathtt{Acq}_{\mathrm{NL}}(l)$, it synchronises with $\mathtt{Rel}_{\mathrm{NL}}(l)$ in thread 1 and ensures that $x^2 := 1$ is completed on lock acquisition.

Note that the node lock l protects the accesses towards locations on *node* 2 *only*. Thus, in Fig. 12c, it only guarantees that $x^2 := 1$ is completed but not necessarily $z^4 := 1$ (towards node 4), and thus $(a, c){=}(1, 0)$ is an allowed outcome. By contrast, Fig. 8f in the overview showcases the lock guarantees: as x and y both reside on node 2, the lock ensures that their accesses by threads 1 and 3 are mutually exclusive, i.e. $a \neq b$ is disallowed. Specifically, if thread 1 acquires l first, x and y are modified before being read by thread 3, i.e. $a{=}b{=}1$.

Conversely, if thread 3 acquires l first, x and y are read before being modified by thread 1, i.e. $a{=}b{=}0$.

Definition 7 (NLOCK-consistency). *A lock-well-formed execution* $\mathcal{G} = \langle E, \mathsf{po}, \mathtt{stmp}, \mathsf{so}, \mathsf{hb}\rangle$ *is* NLOCK*-consistent iff:*

1) $\mathtt{stmp} = \mathtt{stmp}_{\mathtt{NL}}$ *(where* $\mathtt{stmp}_{\mathtt{NL}}$ *is as defined above); and*
2) $\mathsf{so} = \left\{\langle\langle \mathsf{e}, \mathtt{aRF}_{\mathtt{n}(\mathtt{loc}(\mathsf{e}))}\rangle, \langle \mathsf{e}, \mathtt{aNRW}_{\mathtt{n}(\mathtt{loc}(\mathsf{e}))}\rangle\rangle \;\middle|\; \mathtt{m}(\mathsf{e}) = \mathtt{Rel}_{\mathrm{NL}}\right\}$
$\bigcup_{x\in\mathsf{Loc}} \left\{\langle\langle \mathsf{e}_1, \mathtt{aNRW}_{\mathtt{n}(\mathtt{loc}(\mathsf{e}_1))}\rangle, \langle \mathsf{e}_2, \mathtt{aMF}\rangle\rangle \;\middle|\; \langle \mathsf{e}_1, \mathsf{e}_2\rangle \in (\mathsf{po}_x|_{imm})^{-1}; \mathsf{lo}_x\right\}$
where lo_x *is a total order on* $\{\mathsf{e} \in E_x \mid \mathtt{m}(\mathsf{e}) = \mathtt{Acq}_{\mathrm{NL}}\}$.

The NLOCK Implementation. We implement NLOCK as a centralised ticket lock using remote RMWs. For each (node) lock x associated with node $\mathtt{n}(x)$, we create two remote locations x_a and x_r on $\mathtt{n}(x)$. As before, x_a is the ticket dispenser and records the next available ticket. The x_r tracks the release counter and indicates which ticket currently holds the lock. Each thread also uses a local location p_x^t to hold the result of remote operations.

$I_{\mathtt{NL}}(t, \mathtt{Acq}_{\mathrm{NL}}, (x)) \triangleq$
  $\mathtt{RFAA}(p_x^t, x_a, 1, d); \mathtt{Wait}(d);$
  $\mathtt{let}\ v = \mathtt{Read}(p_x^t)\ \mathtt{in}$
  $\mathtt{loop}\ \{$
    $\mathtt{Get}(p_x^t, x_r, d); \mathtt{Wait}(d);$
    $\mathtt{if}\ \mathtt{Read}(p_x^t) = v\ \mathtt{then\ break}\ \};$
  $\mathtt{Write}(p_x^t, v+1)$

$I_{\mathtt{NL}}(t, \mathtt{Rel}_{\mathrm{NL}}, (x)) \triangleq$
  $\mathtt{Rfence}(\mathtt{n}(x));$
  $\mathtt{Put}(x_r, p_x^t, \_)$

Fig. 13: Node lock implementation ($I_{\mathtt{NL}}$) using $\mathrm{RDMA}^{\mathrm{WAIT}}_{\mathrm{RMW}}$

Acquiring the lock on x calls a fetch-and-add on x_a to fetch the next available ticket in p_x^t and increments x_a. It then records the ticket value in v and repeatedly examines x_r until it has value v, indicating that its turn has come and thus t holds the lock. Finally, it increments its ticket value in p_x^t in preparation for later releasing the lock; i.e. p_x^t now records the ticket whose turn is next. As such, releasing the lock simply updates x_r to p_x^t using a `Put` rather than an RMW; this is because only the lock holder can write to x_r. Note that the preceding `Rfence` ensures that earlier `Get` operations towards $\mathtt{n}(x)$ have completed before the lock is release. We prove (Theorem 3) that our implementation is correct against the NLOCK specification with the full proof given in §B.4.

Theorem 3. *The implementation* $I_{\mathtt{NL}}$ *is sound.*

5 The $\mathrm{RDMA}^{\mathrm{SC}}_{\mathrm{RMW}}$ Library

We specify (§5.1), implement, and verify (§5.2) the $\mathrm{RDMA}^{\mathrm{SC}}_{\mathrm{RMW}}$ library that provides *intuitive* read, write, and RMW operations with the strong semantics of *sequential consistency* (SC) [23]. That is, as with SC, the instructions in each thread under $\mathrm{RDMA}^{\mathrm{SC}}_{\mathrm{RMW}}$ are always observed in (program) order. Moreover, unlike in $\mathrm{RDMA}^{\mathrm{WAIT}}_{\mathrm{RMW}}$ or the lock libraries in §4, the users do not need to specify whether a location is local or remote and which node it resides on. For instance, a user can simply call $\mathtt{Write}_{\mathrm{SC}}(\mathtt{x}, v)$ to write (with SC semantics) to location `x`,

regardless of whether $\mathtt{x}$ is local (on the current node) or remote. As such, we use the `typewriter` font and write $\mathtt{x}$ to denote an abstract $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ location whose underlying memory address may be local (i.e. $\mathtt{x}{=}x$) or on a remote node n (i.e. $\mathtt{x}{=}x^n$).

5.1 The $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ Specification

The $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ Methods. The $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ library has four methods: $\mathtt{Read}_{\text{SC}}(\mathtt{x})$, to read from $\mathtt{x}$; $\mathtt{Write}_{\text{SC}}(\mathtt{x}, v)$ to write v to $\mathtt{x}$; $\mathtt{CAS}_{\text{SC}}(\mathtt{x}, v_1, v_2)$, a compare-and-swap on $\mathtt{x}$; and $\mathtt{FAA}_{\text{SC}}(\mathtt{x}, v)$, a fetch-and-add on $\mathtt{x}$. We define $\mathtt{loc}$ as expected, i.e. $\mathtt{loc}(\mathtt{Write}_{\text{SC}}(\mathtt{x}, v)) = \mathtt{loc}(\mathtt{Read}_{\text{SC}}(\mathtt{x})) = \mathtt{loc}(\mathtt{CAS}_{\text{SC}}(\mathtt{x}, v_1, v_2)) = \mathtt{loc}(\mathtt{FAA}_{\text{SC}}(\mathtt{x}, v)) = \{\mathtt{x}\}$. We extend po and $\mathtt{loc}$ to subevents as expected.

Well-formedness. Given an $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ execution $\mathcal{G}$, we define the sets of read subevents ($\mathcal{R}$) to comprise all subevents except writes and the set of write subevents ($\mathcal{W}$) to include all subevents except reads and failed RMWs.

$$\mathcal{R} \triangleq \{\langle \mathsf{e}, \mathtt{aMF}\rangle \mid \mathsf{e} \in \mathcal{G}.E \setminus \{\langle \_, \_, \langle \mathtt{Write}_{\text{SC}}, \_, \_\rangle\rangle\}\}$$

$$\mathcal{W} \triangleq \{\langle \mathsf{e}, \mathtt{aMF}\rangle \mid \mathsf{e} \in \mathcal{G}.E \setminus \{\langle \_, \_, \langle \mathtt{Read}_{\text{SC}}, \_, \_\rangle\rangle\} \setminus \{\langle \_, \_, \langle \mathtt{CAS}_{\text{SC}}, (\_, v, \_), v'\rangle\rangle \mid v \neq v'\}\}$$

As before, a tuple $\langle \mathsf{v_R}, \mathsf{v_W}, \mathsf{rf}, \mathsf{mo}\rangle$ is *well-formed* if the following holds:

- $\mathsf{v_R}/\mathsf{v_W}$ map each read/write subevent to the value read/written:

$$\mathsf{v_R}(\langle\langle \_, \_, \langle \_, \_, v\rangle\rangle, \_\rangle) \triangleq v \qquad \mathsf{v_W}(\langle\langle \_, \_, \langle \mathtt{CAS}_{\text{SC}}, (\_, v_1, v_2), v_1\rangle\rangle, \_\rangle) \triangleq v_2$$

$$\mathsf{v_W}(\langle\langle \_, \_, \langle \mathtt{Write}_{\text{SC}}, (\_, v), \_\rangle\rangle, \_\rangle) \triangleq v \qquad \mathsf{v_W}(\langle\langle \_, \_, \langle \mathtt{FAA}_{\text{SC}}, (\_, v), v'\rangle\rangle, \_\rangle) \triangleq v{+}v'$$

- rf and mo satisfy the same constraints as well-formedness of $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ (§3.2).

We next define $\text{RDMA}^{\text{SC}}_{\text{RMW}}$-consistency, which requires that 1) each event be associated with (single) stamp $\mathtt{aMF}$; and 2) $\mathsf{so} = \mathsf{po} \cup \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb}$. The former ensures that $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ calls remain ordered with respect to other non-RDMA operations. The latter captures the standard notion of happens-before in SC [27].

Definition 8 ($\text{RDMA}^{\text{SC}}_{\text{RMW}}$-consistency). *Execution $\mathcal{G}$ is $\text{RDMA}^{\text{SC}}_{\text{RMW}}$-consistent if:*

1) $\forall \mathsf{e} \in E.\ \mathtt{stmp}(e) = \{\mathtt{aMF}\}$, *and*
2) there exists a well-formed $\langle \mathsf{v_R}, \mathsf{v_W}, \mathsf{rf}, \mathsf{mo}\rangle$ *such that* $\mathcal{G}.\mathsf{so} = \mathcal{G}.\mathsf{po} \cup \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb}$, *where* rb *is defined as in §3.2.*

5.2 The $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ Implementation

We implement $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ using node locks and $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ operations, as shown in Fig. 14. For each $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ location $\mathtt{x}$, we create an $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ location x on some arbitrary node. We assume each thread t has access to a private location r_t for recording the remote data it reads, and a private location $p^t_{\mathtt{x}}$ for recording

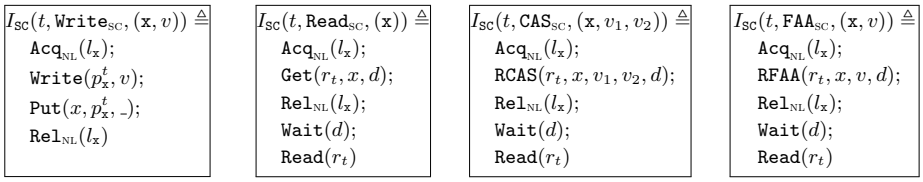
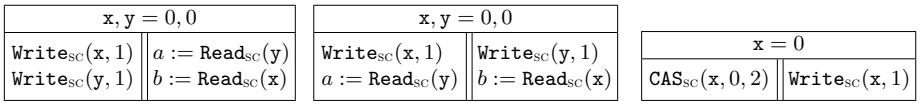
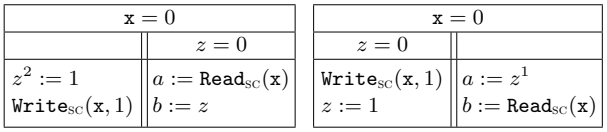


Fig. 14: The implementation of $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ (through the $I_{\mathtt{sc}}$ function)



(a) $(a, b) = (1, 0)$ ✗ (b) $(a, b) = (0, 0)$ ✗ (c) $\mathtt{x} = 2$ ✗



(d) $(a, b) = (1, 0)$ ✓ (e) $(a, b) = (1, 0)$ ✓

Fig. 15: $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ examples

the value to be put to a remote location (i.e. the second argument of a $\mathtt{Put}$)[3]. Moreover, each location $\mathtt{x}$ is associated with a node lock $l_{\mathtt{x}}$ hosted on the same node as x. We implement $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ writes, reads, and RMWs respectively using $\mathtt{Put}$, $\mathtt{Get}$, and remote RMWs of $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ while holding the $l_{\mathtt{x}}$ lock.

Note that the $\mathtt{Write}_{\mathrm{SC}}(\mathtt{x}, v)$ implementation does not wait for $\mathtt{Put}(x, p^t_{\mathtt{x}}, \_)$ to complete. As such, when running $\mathtt{Write}_{\mathrm{SC}}(\mathtt{x}, 1); \mathtt{Write}_{\mathrm{SC}}(\mathtt{y}, 1)$ in Fig. 15a with $\mathtt{n}(x) \neq \mathtt{n}(y)$, location y may be modified before x. However, this out-of-order completion is *not observable*, i.e. the (non-SC) outcome (a, b)=$(1, 0)$ is disallowed, because re-acquiring a node lock makes all previous operations towards its node visible (see §4.4). Specifically, a=1 implies that $\mathtt{Put}(x, p^t_{\mathtt{x}}, \_)$ has been issued. As the implementation of $\mathtt{Read}_{\mathrm{SC}}(\mathtt{x})$ acquires $l_{\mathtt{x}}$, this enforces $\mathtt{Put}(x, p^t_{\mathtt{x}}, \_)$ to become visible; i.e. $\mathtt{Read}_{\mathrm{SC}}(\mathtt{x})$ reads 1 and (a, b)=$(1, 0)$ is disallowed.

In contrast to $\mathtt{Write}_{\mathrm{SC}}(\mathtt{x}, v)$, the implementations of the other three operations must wait (via $\mathtt{Wait}(d)$) for their remote operations to complete prior to reading the result via $\mathtt{Read}(r_t)$ to ensure they observe the correct value. For instance, were we to remove $\mathtt{Wait}(d)$ in the implementation of $\mathtt{Read}_{\mathrm{SC}}(\mathtt{x})$, the $\mathtt{Read}(r_t)$ could read a stale value from r_t *before* $\mathtt{Get}(r_t, x, d)$ completes and updates r_t. Nevertheless, it is sufficient to wait for the remote operation to complete *after* releasing the lock. That is, it is possible for another thread to acquire $l_{\mathtt{x}}$ (and modify x) before $\mathtt{Get}(r_t, x, d)$ completes; however, the semantics of NLOCK ensures that $\mathtt{Get}(r_t, x, d)$ reads the old value into r_t.

The $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ library, when used in isolation (without calls to e.g. $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$), ensures SC behaviour. As such, the weak behaviours of 'message-passing' in

[3] In practice, we can use a $\mathtt{Put}$ with 'inlined data' and forgo temporary location $p^t_{\mathtt{x}}$.

Fig. 15a and 'store-buffering' in Fig. 15b are disallowed. Moreover, $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ RMW operations are strongly isolated with $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ reads and writes; e.g. outcome x=2 is disallowed in Fig. 15c. This is in contrast to remote RMWs of $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$, where outcome x=2 is allowed in Fig. 6b. However, $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ operations does not ensure that earlier remote operations by *other libraries* are completed, and thus outcome (a, b)=$(1, 0)$ is allowed in both Figs. 15d and 15e.

More generally, we can use this strategy to *linearise* [19] accesses to any sequential data structure D by wrapping each call to D inside a node lock. This allows us to port existing sequential data structures to RDMA settings with minimal effort. Finally, we prove (Theorem 4) that our implementation is correct against the $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ specification with the full proof given in §B.5.

Theorem 4. *The implementation I_{sc} is sound.*

6 Related Work

RDMA Semantics. The coreRMA model [13] is an early attempt at formalising remote memory accesses, but this semantics does not match the RDMA technical specification. This gap is addressed by RDMA^{TSO} [3], which formalises the actual RDMA semantics over TSO, but the formalisation did not cover remote RMWs. A later model, RDMA^{SC} [5], explored the semantics from RDMA^{TSO} [3] but over an SC CPU alongside programming strategies to efficiently prevent weak behaviours. RDMA^{SC} is unrelated to our work, including $\text{RDMA}^{\text{SC}}_{\text{RMW}}$.

RDMA-Based Distributed Systems. Besides LOCO [4, 20], prior work has covered a range of distributed systems, e.g. consensus protocols [1], databases [2, 24], stand-alone data structures [9, 14]. However, unlike LOCO (and our work), these are bespoke systems rather than a programming methodology or library.

Verification. Our proofs for the soundness of library implementations have followed the declarative style [4, 27, 31]. For $\text{RDMA}^{\text{TSO}}_{\text{RMW}}$ (like RDMA^{TSO}), we also provide an operational model (§D) which could ultimately form a basis for a program logic (e.g., [7, 22]), ultimately enabling operational abstractions and proofs of refinement [12, 30]. We consider such extensions to be future work.

RDMA Locks. There are several implementations of network locks using RDMA operations, including centralised lock managers [11], decentralised algorithms [33], asymmetric implementations to favour local accesses [6], and technology-agnostic designs that are more general than RDMA [15]. Other stated objectives of these implementations can include fairness, starvation freedom, low latency, load balancing, scalability, contention mitigation, fault tolerance [18], etc.

However, none of these existing implementations have been formally verified (since $\text{RDMA}^{\text{TSO}}_{\text{RMW}}$ is the first formal semantics of remote RMWs). These works, at most, have offered intuitive explanations to support the correctness of their approach. Moreover, these implementations lack an explicit description of the interaction guarantees between locks and other RDMA operations, which as we have seen can be subtle. In most cases, programmers are made responsible to ensure relevant operations are completed before releasing the lock, thus aligning with the weak lock semantics that we have presented.

References

1. Aguilera, M.K., Ben-David, N., Guerraoui, R., Marathe, V.J., Xygkis, A., Zablotchi, I.: Microsecond consensus for microsecond applications. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). pp. 599–616. USENIX Association (Nov 2020), https://www.usenix.org/conference/osdi20/presentation/aguilera
2. Alquraan, A., Udayashankar, S., Marathe, V., Wong, B., Al-Kiswany, S.: 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 (2024)
3. Ambal, G., Dongol, B., Eran, H., Klimis, V., Lahav, O., Raad, A.: Semantics of remote direct memory access: Operational and declarative models of RDMA on TSO architectures. Proc. ACM Program. Lang. **8**(OOPSLA2), 1982–2009 (2024). https://doi.org/10.1145/3689781, https://doi.org/10.1145/3689781
4. Ambal, G., Hodgkins, G., Madler, M., Chockler, G., Dongol, B., Izraelevitz, J., Raad, A., Vafeiadis, V.: A verified high-performance composable object library for remote direct memory access (extended version) (2025), https://arxiv.org/abs/2510.10531
5. Ambal, G., Lahav, O., Raad, A.: Sufficient conditions for robustness of RDMA programs. In: Vafeiadis, V. (ed.) Programming Languages and Systems - 34th European Symposium on Programming, ESOP 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part I. Lecture Notes in Computer Science, vol. 15694, pp. 56–87. Springer (2025). https://doi.org/10.1007/978-3-031-91118-7_3, https://doi.org/10.1007/978-3-031-91118-7_3
6. Baran, A., Nelson-Slivon, J., Tseng, L., Palmieri, R.: Alock: Asymmetric lock primitive for rdma systems. In: Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures. p. 15–26. SPAA '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3626183.3659977, https://doi.org/10.1145/3626183.3659977
7. Bila, E.V., Dongol, B., Lahav, O., Raad, A., Wickerson, J.: View-based owicki–gries reasoning for persistent x86-tso. In: Sergey, I. (ed.) Programming Languages and Systems. pp. 234–261. Springer International Publishing, Cham (2022)
8. Blundell, C., Lewis, E.C., Martin, M.M.: Subtleties of transactional memory atomicity semantics. IEEE Computer Architecture Letters **5**(2), 17–17 (2006). https://doi.org/10.1109/L-CA.2006.18
9. Brock, B., Buluç, A., Yelick, K.: 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 (2019). https://doi.org/10.1145/3337821.3337912, https://doi.org/10.1145/3337821.3337912
10. Chong, N., Sorensen, T., Wickerson, J.: The semantics of transactions and weak memory in x86, power, arm, and C++. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 211–225. ACM (2018). https://doi.org/10.1145/3192366.3192373, https://doi.org/10.1145/3192366.3192373

11. Chung, Y., Zamanian, E.: Using RDMA for lock management. CoRR **abs/1507.03274** (2015), http://arxiv.org/abs/1507.03274
12. Dalvandi, S., Dongol, B.: Implementing and verifying release-acquire transactional memory in C11. Proc. ACM Program. Lang. **6**(OOPSLA2), 1817–1844 (2022). https://doi.org/10.1145/3563352, https://doi.org/10.1145/3563352
13. Dan, A.M., Lam, P., Hoefler, T., Vechev, M.: Modeling and analysis of remote memory access programming. SIGPLAN Not. **51**(10), 129–144 (oct 2016). https://doi.org/10.1145/3022671.2984033, https://doi.org/10.1145/3022671.2984033
14. Devarajan, H., Kougkas, A., Bateman, K., Sun, X.H.: Hcl: Distributing parallel data structures in extreme scales. In: 2020 IEEE International Conference on Cluster Computing (CLUSTER). pp. 248–258 (2020). https://doi.org/10.1109/CLUSTER49012.2020.00035
15. Devulapalli, A., Wyckoff, P.: Distributed queue-based locking using advanced network features. In: 34th International Conference on Parallel Processing (ICPP 2005), 14-17 June 2005, Oslo, Norway. pp. 408–415. IEEE Computer Society (2005). https://doi.org/10.1109/ICPP.2005.34, https://doi.org/10.1109/ICPP.2005.34
16. Dongol, B., Jagadeesan, R., Riely, J.: Transactions in relaxed memory architectures. Proc. ACM Program. Lang. **2**(POPL), 18:1–18:29 (2018). https://doi.org/10.1145/3158106, https://doi.org/10.1145/3158106
17. Gangidi, A., Miao, R., Zheng, S., Bondu, S.J., Goes, G., Morsy, H., Puri, R., Riftadi, M., Shetty, A.J., Yang, J., et al.: Rdma over ethernet for distributed training at meta scale. In: Proceedings of the ACM SIGCOMM 2024 Conference. pp. 57–70 (2024)
18. Gao, J., Wang, Q., Shu, J.: Shiftlock: Mitigate one-sided RDMA lock contention via handover. In: Gunawi, H.S., Tarasov, V. (eds.) 23rd USENIX Conference on File and Storage Technologies, FAST 2025, Santa Clara, CA, February 25-27, 2025. pp. 355–372. USENIX Association (2025), https://www.usenix.org/conference/fast25/presentation/gao
19. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990). https://doi.org/10.1145/78969.78972, https://doi.org/10.1145/78969.78972
20. Hodgkins, G., Madler, M., Izraelevitz, J.: Loco: Rethinking objects for network memory (2025), https://arxiv.org/abs/2503.19270
21. IBTA: Infiniband architecture specification volume 1 release 1.6. https://www.infinibandta.org/ibta-specification/ (2022)
22. Lahav, O., Dongol, B., Wehrheim, H.: Rely-guarantee reasoning for causally consistent shared memory. In: Enea, C., Lal, A. (eds.) Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13964, pp. 206–229. Springer (2023). https://doi.org/10.1007/978-3-031-37706-8_11, https://doi.org/10.1007/978-3-031-37706-8_11
23. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9), 690–691 (Sep 1979). https://doi.org/10.1109/TC.1979.1675439, http://dx.doi.org/10.1109/TC.1979.1675439
24. Li, P., Hua, Y., Zuo, P., Chen, Z., Sheng, J.: ROLEX: A scalable RDMA-oriented learned Key-Value store for disaggregated memory systems. In: 21st USENIX Conference on File and Storage Technologies (FAST 23). pp. 99–114. USENIX Associa-

tion, Santa Clara, CA (Feb 2023), https://www.usenix.org/conference/fast23/presentation/li-pengfei
25. Lu, Y., Chen, G., Li, B., Tan, K., Xiong, Y., Cheng, P., Zhang, J., Chen, E., Moscibroda, T.: {Multi-Path} transport for {RDMA} in datacenters. In: 15th USENIX symposium on networked systems design and implementation (NSDI 18). pp. 357–371 (2018)
26. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-tso. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5674, pp. 391–407. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_27, https://doi.org/10.1007/978-3-642-03359-9_27
27. Raad, A., Doko, M., Rozic, L., Lahav, O., Vafeiadis, V.: On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models. Proc. ACM Program. Lang. **3**(POPL), 68:1–68:31 (2019). https://doi.org/10.1145/3290381, https://doi.org/10.1145/3290381
28. Raad, A., Lahav, O., Vafeiadis, V.: On parallel snapshot isolation and release/acquire consistency. In: Ahmed, A. (ed.) Programming Languages and Systems. pp. 940–967. Springer International Publishing, Cham (2018)
29. Raad, A., Lahav, O., Vafeiadis, V.: On the semantics of snapshot isolation. In: Enea, C., Piskac, R. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 1–23. Springer International Publishing, Cham (2019)
30. Singh, A.K., Lahav, O.: An operational approach to library abstraction under relaxed memory concurrency. Proc. ACM Program. Lang. **7**(POPL), 1542–1572 (2023). https://doi.org/10.1145/3571246, https://doi.org/10.1145/3571246
31. Stefanesco, L., Raad, A., Vafeiadis, V.: Specifying and verifying persistent libraries. In: Weirich, S. (ed.) 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, vol. 14577, pp. 185–211. Springer (2024). https://doi.org/10.1007/978-3-031-57267-8_8, https://doi.org/10.1007/978-3-031-57267-8_8
32. Wang, Z., Luo, L., Ning, Q., Zeng, C., Li, W., Wan, X., Xie, P., Feng, T., Cheng, K., Geng, X., et al.: {SRNIC}: A scalable architecture for {RDMA}{NICs}. In: 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). pp. 1–14 (2023)
33. Yoon, D.Y., Chowdhury, M., Mozafari, B.: Distributed lock management with RDMA: decentralization without starvation. In: Das, G., Jermaine, C.M., Bernstein, P.A. (eds.) Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018. pp. 1571–1586. ACM (2018). https://doi.org/10.1145/3183713.3196890, https://doi.org/10.1145/3183713.3196890
34. Zhu, Y., Eran, H., Firestone, D., Guo, C., Lipshteyn, M., Liron, Y., Padhye, J., Raindel, S., Yahia, M.H., Zhang, M.: Congestion control for large-scale rdma deployments. ACM SIGCOMM Computer Communication Review **45**(4), 523–536 (2015)

A SV Library Semantics

As per other 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 5 methods:

$$\begin{aligned} m(\widetilde{v}) ::= \ & \mathtt{Write}_{\mathrm{SV}}(x, v) \mid \mathtt{Read}_{\mathrm{SV}}(x) \mid \mathtt{Bcast}_{\mathrm{SV}}(x, d, \{n_1, \ldots, n_k\}) \\ & \mid \mathtt{Wait}_{\mathrm{SV}}(d) \mid \mathtt{GFence}(\{n_1, \ldots, n_k\}) \end{aligned}$$

- $\mathtt{Write}_{\mathrm{SV}} : \mathsf{Loc} \times \mathsf{Val} \to ()$
- $\mathtt{Read}_{\mathrm{SV}} : \mathsf{Loc} \to \mathsf{Val}$
- $\mathtt{Wait}_{\mathrm{SV}} : \mathsf{Wid} \to ()$
- $\mathtt{GFence} : \mathcal{P}(\mathsf{Node}) \to ()$

$\mathtt{Write}_{\mathrm{SV}}(x, v)$ 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 is the case for $\mathtt{Put}$, 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 $\mathtt{GFence}(\{n_1, \ldots, n_k\})$ ensures every previous operation of the thread towards one of the nodes in the argument is fully finished, including the writing part of broadcasts.

We then require the stamping function $\mathtt{stmp}_{\mathtt{SV}}$:

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

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

With this, the stamp order is enough to enforce the behaviour of the global fence. If we have a program $\mathtt{Bcast}_{\mathrm{SV}}(x, d, \{\ldots, n, \ldots\}); \mathtt{GFence}(\{\ldots, n, \ldots\})$, the plain execution has two events $\mathtt{e}_{\mathtt{BR}}$ and $\mathtt{e}_{\mathtt{GF}}$, and the definitions of $\mathtt{stmp}_{\mathtt{SV}}$ and sto (cell G12 in Fig. 9) imply $\langle \mathtt{e}_{\mathtt{BR}}, \mathtt{aNRW}_n \rangle \xrightarrow{\mathsf{ppo}} \langle \mathtt{e}_{\mathtt{GF}}, \mathtt{aGF}_n \rangle$.

Recall that, for an execution $\mathcal{G}$, $\mathcal{G}.\mathcal{W}$ represents write subevents (stamps $\mathtt{aCW}$, $\mathtt{aCAS}$, $\mathtt{aNLW}$, and $\mathtt{aNRW}$), while $\mathcal{G}.\mathcal{R}$ represents read subevents (stamps $\mathtt{aCR}$, $\mathtt{aCAS}$, $\mathtt{aNLR}$, $\mathtt{aNAR}$, and $\mathtt{aNRR}$). Recall also that we note e.g. $\mathcal{G}.\mathcal{W}_x \triangleq \{\mathtt{s} \in \mathcal{G}.\mathcal{W} \mid \mathtt{loc}(\mathtt{s}) = \{x\}\}$ to constrain a set to subevents on a specific location x. For the SV library, we additionally define $\mathcal{G}.\mathcal{W}^n \triangleq \{\langle \mathtt{e}, \mathtt{aCW} \rangle \mid \mathtt{n}(\mathtt{t}(\mathtt{e})) = n\} \cup \mathcal{G}.\mathtt{aNRW}_n$ 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 $\mathcal{G}.\mathcal{W}_x^n \triangleq \mathcal{G}.\mathcal{W}_x \cap \mathcal{G}.\mathcal{W}^n$

as expected. Similarly, $\mathcal{G}.\mathcal{R}^n \triangleq \{\mathtt{s} \mid \mathtt{s} \in \mathcal{G}.\mathcal{R} \wedge \mathtt{n}(\mathtt{t}(\mathtt{s})) = n\}$ covers reads occurring on n, either by a CPU read or as part of a broadcast. We now work towards a definition of consistency for shared variables.

Definition 9. *For an execution* $\mathcal{G} = \langle E, \mathsf{po}, \mathtt{stmp}_{\mathtt{SV}}, \_, \_\rangle$*, we define the following:*

- *The* value-read *function* $\mathtt{v}_{\mathtt{R}} : \mathcal{G}.\mathcal{R} \to \mathsf{Val}$ *that associates each read subevent with the value returned, if available, i.e. if* $\mathsf{e} = \langle \_, \_, \langle \mathtt{Read}_{\mathrm{SV}}, \_, v\rangle\rangle$*, then* $\mathtt{v}_{\mathtt{R}}(\mathsf{e}) = v$*.*
- *The* value-written *function* $\mathtt{v}_{\mathtt{W}} : \mathcal{G}.\mathcal{W} \to \mathsf{Val}$ *that associates each write subevent with a value* $\mathcal{G}$*, i.e. if* $\mathsf{e} = \langle \_, \_, \langle \mathtt{Write}_{\mathrm{SV}}, (\_, v), \_\rangle\rangle$*, then* $\mathtt{v}_{\mathtt{W}}(\mathsf{e}) = v$*.*
- *A* reads-from *relation,* $\mathsf{rf} \triangleq \bigcup_n \mathsf{rf}^n$*, where each* $\mathsf{rf}^n \subseteq \mathcal{G}.\mathcal{W}^n \times \mathcal{G}.\mathcal{R}^n$ *is a relation on subevents of the same location and node with matching values, i.e. if* $\langle \mathtt{s}_1, \mathtt{s}_2\rangle \in \mathsf{rf}^n$ *then* $\mathtt{loc}(\mathtt{s}_1) = \mathtt{loc}(\mathtt{s}_2)$ *and* $\mathtt{v}_{\mathtt{W}}(\mathtt{s}_1) = \mathtt{v}_{\mathtt{R}}(\mathtt{s}_2)$*.*
- *A* modification-order *relation* $\mathsf{mo} \triangleq \bigcup_{x,n} \mathsf{mo}^n_x$ *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 $\mathtt{s}_1 = \langle \mathsf{e}, \mathtt{aNLR}_n\rangle \in \mathcal{G}.\mathsf{SEvent}$ and $\mathtt{s}_2 = \langle \mathsf{e}, \mathtt{aNRW}_n\rangle \in \mathcal{G}.\mathsf{SEvent}$, then $\mathtt{v}_{\mathtt{R}}(\mathtt{s}_1) = \mathtt{v}_{\mathtt{W}}(\mathtt{s}_2)$. Each rf^n is functional on its range, i.e. every read in $\mathcal{G}.\mathcal{R}^n$ is related to at most one write in $\mathcal{G}.\mathcal{W}^n$. If a read is not related to a write, it reads the initial value of zero, i.e. if $\mathtt{s}_2 \in \mathcal{G}.\mathcal{R}^n$ and $\mathtt{s}_2 \notin \mathtt{img}(\mathsf{rf}^n)$ then $\mathtt{v}_{\mathtt{R}}(\mathtt{s}_2) = 0$. Finally, each mo^n_x is a strict total order on $\mathcal{G}.\mathcal{W}^n_x$.

We further define the *reads-from-internal* relation as $\mathsf{rf_i} \triangleq [\mathtt{aCW}]; (\mathsf{po} \cap \mathsf{rf}); [\mathtt{aCR}]$ (which corresponds to CPU reads and writes using the same TSO store buffer), and the *reads-from-external* relation as $\mathsf{rf_e} \triangleq \mathsf{rf} \setminus \mathsf{rf_i}$. Moreover, given an execution $\mathcal{G}$ and well-formed rf and mo, we derive additional relations.

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

$$\mathsf{pf} \triangleq \{\langle\langle \mathsf{e}_1, \mathtt{aNLR}_n\rangle, \langle \mathsf{e}_2, \mathtt{aWT}\rangle\rangle \mid \exists d.\ \langle \mathsf{e}_1, \mathsf{e}_2\rangle \in \mathsf{po}|_d\} \qquad \mathsf{rb} \triangleq \bigcup_n \mathsf{rb}^n$$

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

The *polls-from* relation pf states that a $\mathtt{Wait}_{\mathrm{SV}}$ 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 $\textsc{sv}.\mathcal{C}$ as follows.

Definition 10 (SV-consistency). $\langle E, \mathsf{po}, \mathtt{stmp}, \mathsf{so}, \mathsf{hb}\rangle$ *is* SV*-consistent if:*

- $\mathtt{stmp} = \mathtt{stmp}_{\mathtt{SV}}$*;*
- *there exists well-formed* $\mathtt{v}_{\mathtt{R}}$*,* $\mathtt{v}_{\mathtt{W}}$*,* rf*, and* mo*, such that* $[\mathtt{aCR}]; (\mathsf{po}^{-1} \cap \mathsf{rb}); [\mathtt{aCW}] = \emptyset$ *and* $\mathsf{so} = \mathsf{iso} \cup \mathsf{rf_e} \cup \mathsf{pf} \cup \mathsf{rb} \cup \mathsf{mo}$*.*

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 $\texttt{Write}_{\text{SV}}(x,1);\texttt{Read}_{\text{SV}}(x)$ must return 1 and cannot return 0, even if the semantics of TSO allows for the read to finish before the write.

B Correctness Proofs

Correctness proofs of the MOWGLI framework can be found in [4]. We recall the main definitions and results in Appendix B.1 before proving the soundness of the weak lock (§B.2), strong lock (§B.3), node lock (§B.4), and $\text{RDMA}^{\text{SC}}_{\text{RMW}}$ libraries (§B.5).

B.1 Background: MOWGLI Definitions and Results

MOWGLI assumes a type Val of values, a type $\mathsf{Loc} \subseteq \mathsf{Val}$ of locations, and a type Method of methods. The syntax of sequential programs is given by the following grammar:

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

MOWGLI assumes top-level concurrency, i.e. there is a fixed set of threads $\mathsf{Tid} \triangleq \{1, 2, \ldots, T\}$, and a concurrent program is given by a tuple $\widetilde{\mathsf{p}} = \langle \mathsf{p}_1, \ldots, \mathsf{p}_T \rangle$, where each thread t corresponds to a program $\mathsf{p}_t \in \mathsf{SeqProg}$.

The semantics of a program is given by an execution, which is a graph over events. Recall that events are defined in Definition 17. The first two components $\langle E, \mathsf{po} \rangle$ of an execution form a *plain execution*:

Definition 11. *We say that $\langle E, \mathsf{po} \rangle$ is a* plain execution *iff $E \subseteq \mathsf{Event}$, $\mathsf{po} \subseteq E \times E$, and $\mathsf{po} = \bigcup_{t \in \mathsf{Tid}} \mathsf{po}|_t$ where every $\mathsf{po}|_t$ (i.e. po restricted to the events of thread t) is a total order.*

We write $\emptyset_G \triangleq \langle \emptyset, \emptyset \rangle$ for the empty execution and $\{\mathsf{e}\}_G \triangleq \langle \{\mathsf{e}\}, \emptyset \rangle$ for the execution with a single event e. Given two executions, $G_1 = \langle E_1, \mathsf{po}_1 \rangle$ and $G_2 = \langle E_2, \mathsf{po}_2 \rangle$, with disjoint sets of events (i.e. $E_1 \cap E_2 = \emptyset$), we define their sequential composition $G_1; G_2$ and parallel composition $G_1 \| G_2$ as follows:

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

The plain semantics of a program p executed by a thread t is given by $[\![\mathsf{p}]\!]_t$, 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.

$$[\![v]\!]_t \triangleq \{\langle\langle v, 0\rangle, \emptyset_G\rangle\} \qquad [\![\texttt{break}_k\ v]\!]_t \triangleq \{\langle\langle v, k\rangle, \emptyset_G\rangle\}$$

$$[\![m(\widetilde{v})]\!]_t \triangleq \{\langle\langle v', 0\rangle, \{\langle t, \iota, \langle m, \widetilde{v}, v'\rangle\rangle\}_G\rangle \mid v' \in \mathsf{Val} \ \wedge \ \iota \in \mathsf{EventId}\}$$

$$\begin{aligned}[\![\texttt{let p f}]\!]_t \triangleq &\{\langle r, G_1; G_2\rangle \mid \langle\langle v, 0\rangle, G_1\rangle \in [\![\mathsf{p}]\!]_t \ \wedge \ \langle r, G_2\rangle \in [\![\mathsf{f}\ v]\!]_t\} \\ &\cup \{\langle\langle v, k\rangle, G_1\rangle \mid \langle\langle v, k\rangle, G_1\rangle \in [\![\mathsf{p}]\!]_t \ \wedge \ k \neq 0\}\end{aligned}$$

$$[\![\texttt{loop p}]\!]_t \triangleq \bigcup_{j \in \mathbb{N}} \left\{\langle\langle v, k\rangle, G_0; \ldots; G_j\rangle \;\middle|\; (\forall 0 \leq i < j.\ \langle\langle \_, 0\rangle, G_i\rangle \in [\![\mathsf{p}]\!]_t) \ \wedge \ \langle\langle v, k+1\rangle, G_j\rangle \in [\![\mathsf{p}]\!]_t\right\}$$

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

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

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.

Then, we can define executions (Def. 1), libraries (§3.1), and consistent executions (Def. 2).

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

$$\texttt{outcome}_\Lambda(\widetilde{\mathsf{p}}) \triangleq \{\widetilde{v} \mid \exists \langle E, \mathsf{po}, \texttt{stmp}, \mathsf{so}, \mathsf{hb}\rangle\ \Lambda\text{-consistent}.\ \langle \widetilde{v}, \langle E, \mathsf{po}\rangle\rangle \in [\![\widetilde{\mathsf{p}}]\!]\}$$

Then, 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 12. *We say that I is* well defined *for a library L using Λ iff for all $t \in \mathsf{Tid}$, $m \in L.M$ and $\widetilde{v} \in \mathsf{Val}^*$, we have:*

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

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

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

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

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

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

As soundness is difficult to prove directly, MOWGLI develops a modular proof technique using an *abstraction function* mapping the 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 14. *Suppose I is a well-defined implementation of a library L using Λ, and that $G = \langle E, \mathsf{po}\rangle$ and $G' = \langle E', \mathsf{po}'\rangle$ are plain executions using methods of Λ and L respectively. We say that a surjective function $f : E \to E'$ abstracts G to G', denoted $\mathtt{abs}^f_{I,L}(G, G')$, 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);*
- *$f(\mathsf{po}) \subseteq (\mathsf{po}')^*$ and $\forall \mathsf{e}_1, \mathsf{e}_2,\ \langle f(\mathsf{e}_1), f(\mathsf{e}_2)\rangle \in \mathsf{po}' \implies \langle \mathsf{e}_1, \mathsf{e}_2\rangle \in \mathsf{po}$; and*
- *if $\mathsf{e}' = \langle t, \iota, \langle m, \widetilde{v}, v'\rangle\rangle \in E'$ then $\langle \langle v', 0\rangle, G|_{f^{-1}(\mathsf{e}')}\rangle \in [\![I(t, m, \widetilde{v})]\!]_t$*

Lemma 1. *Given $\widetilde{\mathsf{p}}$ on library L and a well-defined implementation I of L, if $\langle \widetilde{v}, G\rangle \in [\![\llbracket \widetilde{\mathsf{p}} \rrbracket_I]\!]$ then there exists $\langle \widetilde{v}, G'\rangle \in [\![\widetilde{\mathsf{p}}]\!]$ and f such that $\mathtt{abs}^f_{I,L}(G, G')$.*

Definition 15. *We say that a well defined implementation I of a library L is* locally sound *iff, whenever we have a Λ-consistent execution $\mathcal{G} = \langle E, \mathsf{po}, \mathtt{stmp}, \mathsf{so}, \mathsf{hb}\rangle$ and $\mathtt{abs}^f_{I,L}(\langle E, \mathsf{po}\rangle, \langle E', \mathsf{po}'\rangle)$, then there exists $\mathtt{stmp}'$, so', and a concretisation function $g : \langle E', \mathsf{po}', \mathtt{stmp}'\rangle.\mathsf{SEvent} \to \mathcal{G}.\mathsf{SEvent}$ 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 $\langle a_0, a'\rangle \in \mathsf{sto}$, there exists $\langle \mathsf{e}_1, a_1\rangle \in \mathcal{G}.\mathsf{SEvent}$ such that $f(\mathsf{e}_1) = \mathsf{e}'$, $\langle a_0, a_1\rangle \in \mathsf{sto}$, and $\langle \langle \mathsf{e}_1, a_1\rangle, \langle \mathsf{e}, a\rangle\rangle \in \mathsf{hb}^*$;*
  - *For all a_0 such that $\langle a', a_0\rangle \in \mathsf{sto}$, there exists $\langle \mathsf{e}_2, a_2\rangle \in \mathcal{G}.\mathsf{SEvent}$ such that $f(\mathsf{e}_2) = \mathsf{e}'$, $\langle a_2, a_0\rangle \in \mathsf{sto}$, and $\langle \langle \mathsf{e}, a\rangle, \langle \mathsf{e}_2, a_2\rangle\rangle \in \mathsf{hb}^*$.*
- *$g(\mathsf{so}') \subseteq \mathsf{hb}$;*
- *For all hb' transitive such that $(\mathsf{ppo}' \cup \mathsf{so}')^+ \subseteq \mathsf{hb}'$ and $g(\mathsf{hb}') \subseteq \mathsf{hb}$, we have $\langle E', \mathsf{po}', \mathtt{stmp}', \mathsf{so}', \mathsf{hb}'\rangle \in L.\mathcal{C}$, where $\mathsf{ppo}' \triangleq \langle E', \mathsf{po}', \mathtt{stmp}'\rangle.\mathsf{ppo}$.*

Theorem 5. *If a well-defined implementation is locally sound, then it is sound.*

We show the local soundness of the different implementations given in this paper. Thus, from the theorem above, these implementations are sound.

B.2 WLOCK Library

Theorem 1. *The implementation $I_{\mathtt{WL}}$ is sound.*

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

$\langle E', \mathsf{po}', \mathtt{stmp}' \rangle.\mathsf{SEvent} \to \mathcal{G}.\mathsf{SEvent}$ respecting some conditions. From $\langle E', \mathsf{po}' \rangle$, we simply take $\mathtt{stmp}' = \mathtt{stmp}_{\mathtt{WL}}$.

Since $\mathcal{G}$ is $\{\text{RDMA}^{\text{WAIT}}_{\text{RMW}}, \text{SV}\}$-consistent, it means $(\mathsf{ppo} \cup \mathsf{so}|_{\text{RDMA}^{\text{WAIT}}_{\text{RMW}}} \cup \mathsf{so}|_{\text{SV}}) \subseteq \mathsf{hb}$, hb is transitive and irreflexive, and the two restrictions of $\mathcal{G}$ are respectively $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$-consistent and SV-consistent.

$\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$-consistency implies there is some well-formed $\mathtt{v}_{\mathtt{R}}$, $\mathtt{v}_{\mathtt{W}}$, rf, mo, nfo, and rao such that ib is irreflexive, $\forall \mathsf{e}.\mathtt{stmp}|_{\text{RDMA}^{\text{WAIT}}_{\text{RMW}}}(\mathsf{e}) \in \mathtt{stmp}_{\mathtt{RW}}(\mathsf{e})$, and $\mathsf{so}|_{\text{RDMA}^{\text{WAIT}}_{\text{RMW}}} = \mathsf{iso} \cup \mathsf{rf}_{\mathsf{e}} \cup \mathsf{pfg} \cup \mathsf{nfo} \cup \mathsf{rb} \cup \mathsf{mo} \cup \mathsf{rao} \cup ([\mathtt{aNRW}_n]; \mathsf{iso}^{-1}; \mathsf{rao}) \cup ([\mathtt{Inst}]; \mathsf{ib})$.

SV-consistency implies there is some well-formed $\mathtt{v}''_{\mathtt{R}}$, $\mathtt{v}''_{\mathtt{W}}$, rf'', and mo'', such that $\mathtt{stmp}|_{\text{SV}} = \mathtt{stmp}_{\mathtt{SV}}$, $[\mathtt{aCR}]; (\mathsf{po}|^{-1}_{\text{SV}} \cap \mathsf{rb}''); [\mathtt{aCW}] = \emptyset$, and $\mathsf{so}|_{\text{SV}} = \mathsf{iso}'' \cup \mathsf{rf}''_{\mathsf{e}} \cup \mathsf{pf}'' \cup \mathsf{rb}'' \cup \mathsf{mo}''$. (We will use double apostrophes for references to the SV library.)

We define g as follows.

- For an event $\mathsf{e}' = (t, \_, (\mathtt{Acq}_{\text{WL}}, (x), ()))$, we choose $g(\mathsf{e}', \mathtt{aMF}) = (\mathsf{e}_r, \mathtt{aCR})$ with $\mathsf{e}_r = (t, \_, (\mathtt{Read}_{\text{SV}}, (x_{t'}), (v))) \in f^{-1}(\mathsf{e}')$ the last event of the implementation (reading a shared variable owned by some thread t').
- For an event $\mathsf{e}' = (t, \_, (\mathtt{Rel}_{\text{WL}}, (x), ()))$, we choose $g(\mathsf{e}', \mathtt{aCW}) = (\mathsf{e}_w, \mathtt{aCW})$ with $\mathsf{e}_w = (t, \_, (\mathtt{Write}_{\text{SV}}, (x_t, v+1), ())) \in f^{-1}(\mathsf{e}')$ the second event of the implementation.

First, let us show that g preserves sto (first property of local soundness). For $\mathtt{Rel}_{\text{WL}}$ this is trivial using the identity function. For $\mathtt{Acq}_{\text{WL}}$, the stamp $\mathtt{aCR}$ is similar to $\mathtt{aMF}$ w.r.t. later stamps, so $(\mathsf{e}_2, a_2) = (\mathsf{e}_r, \mathtt{aCR})$ is enough. For an earlier stamp a_0 such that $(a_0, \mathtt{aMF}) \in \mathsf{sto}$, we take $(\mathsf{e}_1, a_1) = ((t, \_, (\mathtt{RFAA}, (\ldots, d), ())), \mathtt{aNLW}_n)$ the first event of the implementation, and with $\mathsf{e}_{wt} = (t, \_, (\mathtt{Wait}, (d), ()))$ the second event we have $(\mathsf{e}_1, a_1) \xrightarrow{\mathsf{pfg}} (\mathsf{e}_{wt}, \mathtt{aWT}) \xrightarrow{\mathsf{ppo}} (\mathsf{e}_r, \mathtt{aCR})$ (thus included in hb) with $(a_0, \mathtt{aNLW}_n) \in \mathsf{sto}$.

Now we need to pick a suitable so' such that $g(\mathsf{so}') \subseteq \mathsf{hb}$ and $\langle E', \mathsf{po}', \mathtt{stmp}', \mathsf{so}', \_ \rangle$ is WLOCK-consistent. We can assume that $\langle E', \mathsf{po}' \rangle$ respects locks, as otherwise $\mathsf{so}' = \emptyset$ is enough. Thus, for each location x we need to define a total order lo'_x on $A'_x \triangleq \{\mathsf{e}' \mid \mathsf{e}' \in E'_x \wedge \mathtt{m}(\mathsf{e}') = \mathtt{Acq}_{\text{WL}}\}$. Each event $\mathsf{e}' \in A'_x$ can be associated to its first subevent of the form $((t', \_, (\mathtt{RFAA}, (p^{t'}_x, x_a, 1, d), ())), \mathtt{aNAR}_n)$, with $n = \mathtt{n}(x)$. From $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$-consistency, rao induces a total ordering on these subevents, and we simply keep the same ordering for A'_x. As such, we define $\mathsf{so}' = \bigcup_x \left\{ \langle \mathsf{e}'_1, \mathtt{aCW} \rangle, \langle \mathsf{e}'_2, \mathtt{aMF} \rangle \;\middle|\; (\mathsf{e}'_1, \mathsf{e}'_2) \in (\mathsf{po}'_x|_{\text{imm}})^{-1}; \mathsf{lo}'_x \right\}$ as expected, and we have that $\langle E', \mathsf{po}', \mathtt{stmp}', \mathsf{so}', \_ \rangle$ is WLOCK-consistent.

Thus, the rest of the proof is to show that $g(\mathsf{so}') \subseteq \mathsf{hb}$, i.e. that the synchronisations promised by the WLOCK library are enforced in the implementation. We can assume $(\mathsf{e}'_0, \mathtt{aMF}) \xrightarrow{\mathsf{lo}'_x} (\mathsf{e}'_2, \mathtt{aMF})$ and $(\mathsf{e}'_0, \mathtt{aMF}) \xrightarrow{\mathsf{po}'_x|_{\text{imm}}} (\mathsf{e}'_1, \mathtt{aCW})$, with e'_0 running $\mathtt{Acq}_{\text{WL}}(x)$ by thread t_1, e'_1 running $\mathtt{Rel}_{\text{WL}}(x)$ by thread t_1, and e'_2 running $\mathtt{Acq}_{\text{WL}}(x)$ by thread t_2. We also note $(\mathsf{e}_1, \mathtt{aCW}) = g(\mathsf{e}'_1, \mathtt{aCW})$ and $(\mathsf{e}_2, \mathtt{aCR}) = g(\mathsf{e}'_2, \mathtt{aMF})$. Our goal is then to show $(\mathsf{e}_1, \mathtt{aCW}) \xrightarrow{\mathsf{hb}} (\mathsf{e}_2, \mathtt{aCR})$.

We proceed by induction on the ordering lo'_x. The base case is for $(\mathsf{e}'_0, \mathtt{aMF}) \xrightarrow{\mathsf{lo}'_x|_{\text{imm}}} (\mathsf{e}'_2, \mathtt{aMF})$. This base case trivially implies the general case by transitivity, since

the program respects locks (i.e. intermediate acquires are being released) and $(\texttt{aCR}, \texttt{aCW}) \in \mathsf{sto}$.

Let $\mathsf{e}_0^{faa} = (t_1, \_, (\texttt{RFAA}, (p_x^{t_1}, x_a, 1, d), ()))$ be the FAA in the implementation of e'_0 and $\mathsf{e}_2^{faa} = (t_2, \_, (\texttt{RFAA}, (p_x^{t_2}, x_a, 1, d), ()))$ in the implementation of e'_2. By definition we have $(\mathsf{e}_0^{faa}, \texttt{aNAR}_n) \xrightarrow{(\mathsf{rao}|_{E_{x_a}})|_{\mathrm{imm}}} (\mathsf{e}_2^{faa}, \texttt{aNAR}_n)$, since any remote RMW in E_{x_a} is from an implementation of some $\texttt{Acq}_{\mathrm{WL}}(x)$ event. From the semantics of $\mathrm{RDMA}^{\mathrm{WAIT}}_{\mathrm{RMW}}$ we have $(\mathsf{e}_0^{faa}, \texttt{aNRW}_n) \xrightarrow{\mathsf{hb}} (\mathsf{e}_2^{faa}, \texttt{aNAR}_n)$ (from the $([\texttt{aNRW}]; \mathsf{iso}^{-1}; \mathsf{rao})$ component), and thus we necessarily have $(\mathsf{e}_0^{faa}, \texttt{aNRW}_n) \xrightarrow{\mathsf{rf}} (\mathsf{e}_2^{faa}, \texttt{aNAR}_n)$, i.e. the second FAA reads the modified value of the first. This is because e_2^{faa} cannot read from an earlier write (or the initial value of 0) as that would imply an rb dependency and an hb cycle; and cannot read ($\mathsf{rf}_{\mathsf{e}} \subseteq \mathsf{hb}$) from a later write, as any later write is hb after e_2^{faa} (via rao and ppo).

There is some value $v_0 = \mathtt{v}_{\mathtt{R}}((\mathsf{e}_0^{faa}, \texttt{aNAR}_n))$ read by the first FAA operation. By well-formedness of $\mathtt{v}_{\mathtt{R}}$, $\mathtt{v}_{\mathtt{W}}$, and rf, we have $\mathtt{v}_{\mathtt{R}}((\mathsf{e}_2^{faa}, \texttt{aNAR}_n)) = \mathtt{v}_{\mathtt{W}}((\mathsf{e}_0^{faa}, \texttt{aNAR}_n)) = v_0 + 1$, i.e. the following $\texttt{Acq}_{\mathrm{WL}}(x)$ gets the next ticket. More generally, it is clear every $\texttt{Acq}_{\mathrm{WL}}(x)$ gets a different ticket. We also have $\mathtt{v}_{\mathtt{W}}((\mathsf{e}_0^{faa}, \texttt{aNLW}_n)) = v_0$, i.e. $p_x^{t_1}$ is modified to contain v_0. Respectively $p_x^{t_2}$ is modified to contain $v_0 + 1$.

Let e_0^r be the third event of the implementation of e'_0 reading $p_x^{t_1}$. We necessarily have $(\mathsf{e}_0^{faa}, \texttt{aNLW}_n) \xrightarrow{\mathsf{rf}} (\mathsf{e}_0^r, \texttt{aCR})$. This is because e_0^r cannot read from the future (it would create an $\mathsf{rf}; \mathsf{ippo}$ cycle in ib) and the second event $\texttt{Wait}(d)$ makes sure all previous modifications of $p_x^{t_1}$ are available (ignoring the last one would be an $\mathsf{rb}; \mathsf{hb}$ cycle). Thus, in the implementation of e'_0, the meta-variable $\mathtt{v}$ corresponds to the value v_0. More generally, in any implementation of $\texttt{Acq}_{\mathrm{WL}}(x)$, $\mathtt{v}$ corresponds to the ticket obtained (e.g. $v_0 + 1$ for e'_2).

The implementation of e'_1 (running $\texttt{Rel}_{\mathrm{WL}}(x)$) also reads $p_x^{t_1}$. For the same reason, $\mathtt{v}$ corresponds to the ticket of the previous $\texttt{Acq}_{\mathrm{WL}}(x)$, i.e. v_0 in our case. Since the program respects locks, every $\texttt{Rel}_{\mathrm{WL}}(x)$ handles a different ticket, and e'_1 is the only one handling ticket v_0 for x.

The second event in the implementation of e'_1 is $\mathsf{e}_1 = (t_1, \_, (\texttt{Write}_{\mathrm{SV}}, (x_{t_1}, v_0 + 1), ()))$ modifying x_{t_1}. (There is also a broadcast propagating the new value across the network.) By well-formedness we have $\mathtt{v}''_{\mathtt{W}}((\mathsf{e}_1, \texttt{aCW})) = v_0 + 1$. The last event in the implementation of e'_2 is of the form $\mathsf{e}_2 = (t_2, \_, (\texttt{Read}_{\mathrm{SV}}, (x_{t_2}), (v_0 + 1)))$ returning a value of $v_0 + 1$, and by well-formedness $\mathtt{v}''_{\mathtt{R}}(\mathsf{e}_2, \texttt{aCR}) = v_0 + 1$. The read is necessarily on x_{t_2} as other x_t shared variables are never modified to contain $v_0 + 1$. Now, by well-formedness of rf'' we can create a dependency between e_1 and e_2.

If $\mathtt{n}(t_1) = \mathtt{n}(t_2)$ (i.e. the two commands are on the same node, perhaps even the same thread), then we have $(\mathsf{e}_1, \texttt{aCW}) \xrightarrow{\mathsf{rf}''} (\mathsf{e}_2, \texttt{aCR})$ as $(\mathsf{e}_1, \texttt{aCW})$ is the only element of $\mathcal{G}.\mathcal{W}^{\mathtt{n}(t_1)}$ writing $v_0 + 1$. If they are different threads or $\mathsf{e}_2 \xrightarrow{\mathsf{po}} \mathsf{e}_1$, then $\mathsf{rf}''_{\mathsf{e}} \subseteq \mathsf{hb}$ is enough. Otherwise $t_1 = t_2$ with $\mathsf{e}_1 \xrightarrow{\mathsf{po}} \mathsf{e}_2$ and the $\texttt{RFAA}/\texttt{Wait}$ in-between e_1 and e_2 forces a sequence of dependencies $\mathsf{ppo}; \mathsf{pfg}; \mathsf{ppo} \subseteq \mathsf{hb}$.

Else $\mathtt{n}(t_1) \neq \mathtt{n}(t_2)$ and e_2 reads from an element of $\mathcal{G}.\mathcal{W}^{\mathtt{n}(t_2)}$, which is a subevent of a broadcast reading from e_1. (Technically, this could be from

a delayed broadcast of a previous $\texttt{Rel}_{\text{WL}}(x)$ by thread t_1, not necessarily the broadcast immediately after e_1.) Thus we similarly have $((\mathsf{e}_1, \texttt{aCW}), (\mathsf{e}_2, \texttt{aCR})) \in \mathsf{rf}''_{\mathsf{e}}; \mathsf{iso}''; \mathsf{rf}''_{\mathsf{e}} \subseteq \mathsf{hb}$.

B.3 SLOCK Library

$$I_{\texttt{SL}}(t, \texttt{Acq}_{\text{SL}}, (x)) \triangleq \texttt{Acq}_{\text{WL}}(x) \qquad I_{\texttt{SL}}(t, \texttt{Rel}_{\text{SL}}, (x)) \triangleq \texttt{GFence}(\mathsf{Node}); \texttt{Rel}_{\text{WL}}(x)$$

Theorem 2. *The implementation $I_{\texttt{SL}}$ is sound.*

Proof. This is very straightforward from the semantics of the different libraries.

If an execution is lock-well-formed (Definition 4) with respect to strong locks, the implementation is clearly lock-well-formed with respect to weak locks.

A strong acquire $\texttt{Acq}_{\text{SL}}(x)$ should behave as stamp $\texttt{aMF}$, which is the case of the implementation $\texttt{Acq}_{\text{WL}}(x)$. A strong release $\texttt{Rel}_{\text{SL}}(x)$ should behave as a global fence (stamps $\texttt{aGF}_n$) and synchronise with later acquires. In the implementation, the first call executes a global fence (stamps $\texttt{aGF}_n$, see Appendix A), while the latter call is a weak release that synchronises with later acquires (Definition 5). The two components execute in order according to $\mathsf{ppo} \subseteq \mathsf{hb}$ (cell L2 in Fig. 9).

B.4 NLOCK Library

Theorem 3. *The implementation $I_{\texttt{NL}}$ is sound.*

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

Since $\mathcal{G}$ is $\{\text{RDMA}^{\text{WAIT}}_{\text{RMW}}\}$-consistent, it means $(\mathsf{ppo} \cup \mathsf{so}) \subseteq \mathsf{hb}$, hb is transitive and irreflexive, and $\mathcal{G}$ is $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$-consistent. Thus there is some well-formed $\mathsf{v}_{\texttt{R}}$, $\mathsf{v}_{\texttt{W}}$, rf, mo, nfo, and rao such that ib is irreflexive, $\forall \mathsf{e}.\texttt{stmp}(\mathsf{e}) \in \texttt{stmp}_{\texttt{RW}}(\mathsf{e})$, and $\mathsf{so} = \mathsf{iso} \cup \mathsf{rf}_{\mathsf{e}} \cup \mathsf{pfg} \cup \mathsf{nfo} \cup \mathsf{rb} \cup \mathsf{mo} \cup \mathsf{rao} \cup ([\texttt{aNRW}]; \mathsf{iso}^{-1}; \mathsf{rao}) \cup ([\texttt{Inst}]; \mathsf{ib})$.

As an intermediate result: for each thread t we have $\mathsf{mo}_{p^t_x} \subseteq \mathsf{po}$, i.e. the modifications of the temporary location p^t_x happen in program order. Since hb containing mo is acyclic, it is enough to show that whenever $\mathsf{s}_1, \mathsf{s}_2 \in \mathcal{G}.\mathcal{W}_{p^t_x}$ and $(\mathsf{s}_1, \mathsf{s}_2) \in \mathsf{po}$ then we have $(\mathsf{s}_1, \mathsf{s}_2) \in \mathsf{hb}$. If s_1 has a stamp $\texttt{aCW}$ we immediately have $(\mathsf{s}_1, \mathsf{s}_2) \in \mathsf{ppo} \subseteq \mathsf{hb}$. From the implementation, in the other cases s_1 has a stamp $\texttt{aNLW}_n$ from either a $\texttt{RFAA}$ or $\texttt{Get}$ operation. In each case, s_1 is immediately followed by some $(\mathsf{e}, \texttt{aWT})$ forcing the write to finish. Thus we have $(\mathsf{s}_1, \mathsf{s}_2) \in \mathsf{pfg}; \mathsf{ppo} \subseteq \mathsf{hb}$.

We now define g as follows.

- For an event $\mathsf{e}' = (t, \_, (\texttt{Acq}_{\text{NL}}, (x), ()))$, we choose $g(\mathsf{e}', \texttt{aMF}) = (\mathsf{e}_r, \texttt{aCR})$ with $\mathsf{e}_r = (t, \_, (\texttt{Read}, (p^t_x), (v))) \in f^{-1}(\mathsf{e}')$ the last read event before breaking the loop, and penultimate event of the implementation.

- For an event $\mathsf{e}' = (t, \_, (\mathtt{Rel}_{\mathrm{NL}}, (x), ()))$, we choose: $g(\mathsf{e}', \mathtt{aRF}_n) = (\mathsf{e}_{rf}, \mathtt{aRF}_n)$ with $\mathsf{e}_{rf} = (t, \_, (\mathtt{Rfence}, (\mathtt{n}(x)), ())) \in f^{-1}(\mathsf{e}')$ the first event of the implementation; and $g(\mathsf{e}', \mathtt{aNRW}_n) = (\mathsf{e}_{put}, \mathtt{aNRW}_n)$ with $\mathsf{e}_{put} = (t, \_, (\mathtt{Put}, (x_r, p_x^t, \_), ())) \in f^{-1}(\mathsf{e}')$ the second event of the implementation.

First, let us show that g preserves sto (first property of local soundness). For $\mathtt{Rel}_{\mathrm{NL}}$ this is trivial as g maps to the same stamps. For $\mathtt{Acq}_{\mathrm{NL}}$, the stamp $\mathtt{aCR}$ is similar to $\mathtt{aMF}$ w.r.t. later stamps, so $(\mathsf{e}_2, a_2) = (\mathsf{e}_r, \mathtt{aCR})$ is enough. For an earlier stamp a_0 such that $(a_0, \mathtt{aMF}) \in \mathsf{sto}$, we take $(\mathsf{e}_1, a_1) = ((t, \_, (\mathtt{RFAA}, (\ldots, d), ())), \mathtt{aNLW}_n)$ the first event of the implementation, and with $\mathsf{e}_{wt} = (t, \_, (\mathtt{Wait}, (d), ()))$ the second event we have $(\mathsf{e}_1, a_1) \xrightarrow{\mathsf{pfg}} (\mathsf{e}_{wt}, \mathtt{aWT}) \xrightarrow{\mathsf{ppo}} (\mathsf{e}_r, \mathtt{aCR})$ (thus included in hb) with $(a_0, \mathtt{aNLW}_n) \in \mathsf{sto}$.

Now we need to pick a suitable so′ such that $g(\mathsf{so}') \subseteq \mathsf{hb}$ and $\langle E', \mathsf{po}', \mathtt{stmp}', \mathsf{so}', \_\rangle$ is NLOCK-consistent. We can assume that $\langle E', \mathsf{po}'\rangle$ respects locks, as otherwise $\mathsf{so}' = \emptyset$ is enough. Thus, for each location x we need to define a total order lo'_x on $A'_x \triangleq \{\mathsf{e}' \mid \mathsf{e}' \in E'_x \wedge \mathtt{m}(\mathsf{e}') = \mathtt{Acq}_{\mathrm{NL}}\}$. Each event $\mathsf{e}' \in A'_x$ can be associated to its first subevent of the form $((t', \_, (\mathtt{RFAA}, (p_x^{t'}, x_a, 1, d), ())), \mathtt{aNAR}_n)$, with $n = \mathtt{n}(x)$. From $\mathrm{RDMA}^{\mathrm{WAIT}}_{\mathrm{RMW}}$-consistency, rao induces a total ordering on these subevents, and we simply keep the same ordering for A'_x. As such, we define

$$\mathsf{so}' = \left\{\langle \mathsf{e}', \mathtt{aRF}_{\mathtt{n}(\mathtt{loc}(\mathsf{e}'))}\rangle, \langle \mathsf{e}', \mathtt{aNRW}_{\mathtt{n}(\mathtt{loc}(\mathsf{e}'))}\rangle \;\middle|\; \mathtt{m}(\mathsf{e}') = \mathtt{Rel}_{\mathrm{NL}}\right\}$$
$$\bigcup_{x \in \mathsf{Loc}} \left\{\langle \mathsf{e}'_1, \mathtt{aNRW}_{\mathtt{n}(\mathtt{loc}(\mathsf{e}'_1))}\rangle, \langle \mathsf{e}'_2, \mathtt{aMF}\rangle \;\middle|\; (\mathsf{e}'_1, \mathsf{e}'_2) \in (\mathsf{po}'_x|_{\mathrm{imm}})^{-1}; \mathsf{lo}'_x\right\}$$

as expected, and we have that $\langle E', \mathsf{po}', \mathtt{stmp}', \mathsf{so}', \_\rangle$ is NLOCK-consistent.

Thus, the rest of the proof is to show that $g(\mathsf{so}') \subseteq \mathsf{hb}$, i.e. that the synchronisations promised by the NLOCK library are enforced in the implementation. The easy case is for the internal synchronisation. For $(\langle \mathsf{e}', \mathtt{aRF}_n\rangle, \langle \mathsf{e}', \mathtt{aNRW}_n\rangle) \in \mathsf{so}'$, we clearly have $(g(\langle \mathsf{e}', \mathtt{aRF}_n\rangle), g(\langle \mathsf{e}', \mathtt{aNRW}_n\rangle)) \in \mathsf{ppo} \subseteq \mathsf{hb}$.

For the main case, we can assume $(\mathsf{e}'_0, \mathtt{aMF}) \xrightarrow{\mathsf{lo}'_x} (\mathsf{e}'_2, \mathtt{aMF})$ and $(\mathsf{e}'_0, \mathtt{aMF}) \xrightarrow{\mathsf{po}'_x|_{\mathrm{imm}}} (\mathsf{e}'_1, \mathtt{aNRW}_n)$, with $\mathtt{n}(x) = n$, e'_0 running $\mathtt{Acq}_{\mathrm{NL}}(x)$ by thread t_1, e'_1 running $\mathtt{Rel}_{\mathrm{NL}}(x)$ by thread t_1, and e'_2 running $\mathtt{Acq}_{\mathrm{NL}}(x)$ by thread t_2. We also note $(\mathsf{e}_1, \mathtt{aNRW}_n) = g(\mathsf{e}'_1, \mathtt{aNRW}_n)$ and $(\mathsf{e}_2, \mathtt{aCR}) = g(\mathsf{e}'_2, \mathtt{aMF})$. Our goal is then to show $(\mathsf{e}_1, \mathtt{aNRW}_n) \xrightarrow{\mathsf{hb}} (\mathsf{e}_2, \mathtt{aCR})$.

We proceed by induction on the ordering lo'_x. The base case is for $(\mathsf{e}'_0, \mathtt{aMF}) \xrightarrow{\mathsf{lo}'_x|_{\mathrm{imm}}} (\mathsf{e}'_2, \mathtt{aMF})$. This base case trivially implies the general case by transitivity, since the program respects locks (i.e. intermediate acquires are being released) and $(\mathtt{aCR}, \mathtt{aNRW}_n) \in \mathsf{sto}$.

Let $\mathsf{e}_0^{faa} = (t_1, \_, (\mathtt{RFAA}, (p_x^{t_1}, x_a, 1, d), ()))$ be the FAA in the implementation of e'_0 and $\mathsf{e}_2^{faa} = (t_2, \_, (\mathtt{RFAA}, (p_x^{t_2}, x_a, 1, d), ()))$ in the implementation of e'_2. By definition we have $(\mathsf{e}_0^{faa}, \mathtt{aNAR}_n) \xrightarrow{(\mathsf{rao}|_{E_{x_a}})|_{\mathrm{imm}}} (\mathsf{e}_2^{faa}, \mathtt{aNAR}_n)$, since any remote RMW in E_{x_a} is from an implementation of some $\mathtt{Acq}_{\mathrm{WL}}(x)$ event. From the semantics of $\mathrm{RDMA}^{\mathrm{WAIT}}_{\mathrm{RMW}}$ we have $(\mathsf{e}_0^{faa}, \mathtt{aNRW}_n) \xrightarrow{\mathsf{hb}} (\mathsf{e}_2^{faa}, \mathtt{aNAR}_n)$ (from the $([\mathtt{aNRW}]; \mathsf{iso}^{-1}; \mathsf{rao})$ component), and thus we necessarily have $(\mathsf{e}_0^{faa}, \mathtt{aNRW}_n) \xrightarrow{\mathsf{rf}}$

$(\mathtt{e}_2^{faa}, \mathtt{aNAR}_n)$, i.e. the second FAA reads the modified value of the first. This is because $\mathtt{e}_2^{faa}$ cannot read from an earlier write (or the initial value of 0) as that would imply an rb dependency and an hb cycle; and cannot read ($\mathsf{rf_e} \subseteq \mathsf{hb}$) from a later write, as any later write is hb after $\mathtt{e}_2^{faa}$ (via rao and ppo).

There is some value $v_0 = \mathtt{v_R}((\mathtt{e}_0^{faa}, \mathtt{aNAR}_n))$ read by the first FAA operation. By well-formedness of $\mathtt{v_R}$, $\mathtt{v_W}$, and rf, we have $\mathtt{v_R}((\mathtt{e}_2^{faa}, \mathtt{aNAR}_n)) = \mathtt{v_W}((\mathtt{e}_0^{faa}, \mathtt{aNAR}_n)) = v_0 + 1$, i.e. the following $\mathtt{Acq}_{\mathrm{WL}}(x)$ gets the next ticket. More generally, it is clear every $\mathtt{Acq}_{\mathrm{WL}}(x)$ gets a different ticket. We also have $\mathtt{v_W}((\mathtt{e}_0^{faa}, \mathtt{aNLW}_n)) = v_0$, i.e. $p_x^{t_1}$ is modified to contain v_0. Respectively $p_x^{t_2}$ is modified to contain $v_0 + 1$.

Let $\mathtt{e}_0^r$ be the third event of the implementation of $\mathtt{e}_0'$ reading $p_x^{t_1}$. We necessarily have $(\mathtt{e}_0^{faa}, \mathtt{aNLW}_n) \xrightarrow{\mathsf{rf}} (\mathtt{e}_0^r, \mathtt{aCR})$. This is because $\mathtt{e}_0^r$ cannot read from the future (it would create an $\mathsf{rf}; \mathsf{ippo}$ cycle in ib) and the second event $\mathtt{Wait}(d)$ makes sure all previous modifications of $p_x^{t_1}$ are available (ignoring the last one would be an $\mathsf{rb}; \mathsf{hb}$ cycle since $\mathsf{mo}_{p_x^{t_1}} \subseteq \mathsf{po}$). Thus, in the implementation of $\mathtt{e}_0'$, the meta-variable $\mathtt{v}$ corresponds to the value v_0. More generally, in any implementation of $\mathtt{Acq}_{\mathrm{WL}}(x)$, $\mathtt{v}$ corresponds to the ticket obtained (e.g. $v_0 + 1$ for $\mathtt{e}_2'$). So the last event $\mathtt{e}_0^w$ of the implementation of $\mathtt{e}_0'$ modifies $p_x^{t_1}$ to $v_0 + 1$.

The implementation of $\mathtt{e}_1'$ (running $\mathtt{Rel}_{\mathrm{WL}}(x)$) has an operation Put (event $\mathtt{e}_1$) reading $p_x^{t_1}$ to send to x_r. We necessarily have $(\mathtt{e}_0^w, \mathtt{aCW}) \xrightarrow{\mathsf{rf}} (\mathtt{e}_1, \mathtt{aNLR}_n)$, since the write is available ($(\mathtt{aCW}, \mathtt{aNLR}_n) \in \mathsf{sto}$) and later write on $p_x^{t_1}$ from later RFAA are not finished ($(\mathtt{aNLR}_n, \mathtt{aNLW}_n) \in \mathsf{sto}$ and $\mathtt{n}(x_r) = \mathtt{n}(x_a)$). Thus $\mathtt{v_W}((\mathtt{e}_1, \mathtt{aNRW}_n)) = \mathtt{v_R}((\mathtt{e}_1, \mathtt{aNLR}_n)) = v_0 + 1$. More generally, each $\mathtt{Rel}_{\mathrm{WL}}(x)$ modifies x_r to contain the next value after the ticket obtained by the previous $\mathtt{Acq}_{\mathrm{WL}}(x)$ operation. Since each $\mathtt{Acq}_{\mathrm{WL}}(x)$ handles a different ticket, this is the only modification of x_r to contain $v_0 + 1$.

The penultimate event in the implementation of $\mathtt{e}_2'$ (causing the loop break) is of the form $\mathtt{e}_2 = (t_2, \_, (\mathtt{Read}, (p_x^{t_2}), (v_0 + 1)))$ returning a value of $v_0 + 1$, and by well-formedness $\mathtt{v_R}((\mathtt{e}_2, \mathtt{aCR})) = v_0 + 1$. If we note $\mathtt{e}_2^{get} = (t_2, \_, (\mathtt{Get}, (p_x^{t_2}, x_r, d), ()))$ the last Get event preceding $\mathtt{e}_2$ in the implementation, we clearly have $(\mathtt{e}_2^{get}, \mathtt{aNLW}_n) \xrightarrow{\mathsf{rf}} (\mathtt{e}_2, \mathtt{aCR})$ (as previously, the intermediate Wait makes the write available), and $\mathtt{v_R}((\mathtt{e}_2^{get}, \mathtt{aNRR}_n)) = \mathtt{v_W}((\mathtt{e}_2^{get}, \mathtt{aNLW}_n)) = \mathtt{v_R}((\mathtt{e}_2, \mathtt{aCR})) = v_0 + 1$.

By well-formedness of rf, we also have $(\mathtt{e}_1, \mathtt{aNRW}_n) \xrightarrow{\mathsf{rf}} (\mathtt{e}_2^{get}, \mathtt{aNRR}_n)$ from the only write of $v_0 + 1$ on x_r. Finally, we have $((\mathtt{e}_1, \mathtt{aNRW}_n), (\mathtt{e}_2, \mathtt{aCR})) \in \mathsf{rf_e}; \mathsf{iso}; \mathsf{rf_e} \subseteq \mathsf{hb}$.

B.5 $\mathrm{RDMA}^{\mathrm{SC}}_{\mathrm{RMW}}$ Library

Theorem 4. *The implementation I_{sc} is sound.*

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

Since $\mathcal{G}$ is $\{\text{RDMA}^{\text{WAIT}}_{\text{RMW}}, \text{NLOCK}\}$-consistent, it means $(\mathsf{ppo} \cup \mathsf{so}|_{\text{RDMA}^{\text{WAIT}}_{\text{RMW}}} \cup \mathsf{so}|_{\text{NLOCK}}) \subseteq \mathsf{hb}$, hb is transitive and irreflexive, and the two restrictions of $\mathcal{G}$ are respectively $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$-consistent and NLOCK-consistent.

$\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$-consistency implies there is some well-formed $\mathtt{v_R}$, $\mathtt{v_W}$, rf, mo, nfo, and rao such that ib is irreflexive, $\forall \mathsf{e}.\mathtt{stmp}|_{\text{RDMA}^{\text{WAIT}}_{\text{RMW}}}(\mathsf{e}) \in \mathtt{stmp_{RW}}(\mathsf{e})$, and $\mathsf{so}|_{\text{RDMA}^{\text{WAIT}}_{\text{RMW}}} = \mathsf{iso} \cup \mathsf{rf_e} \cup \mathsf{pfg} \cup \mathsf{nfo} \cup \mathsf{rb} \cup \mathsf{mo} \cup \mathsf{rao} \cup ([\mathtt{aNRW}]; \mathsf{iso}^{-1}; \mathsf{rao}) \cup ([\mathtt{Inst}]; \mathsf{ib})$.

$I_{\mathtt{SC}}$ respects locks, as every operation is implemented to contain an $\mathtt{Acq}_{\text{NL}}$ (first) and a $\mathtt{Rel}_{\text{NL}}$ operation (later) on the same lock location. As such $\langle E|_{\text{NLOCK}}, \mathsf{po}|_{\text{NLOCK}}\rangle$ respects locks. So NLOCK-consistency implies $\mathtt{stmp}|_{\text{NLOCK}} = \mathtt{stmp_{NL}}$ and for each lock location l there is a total order lo_l on $\{\mathsf{e} \mid \mathsf{e} \in E_l \wedge \mathtt{m}(\mathsf{e}) = \mathtt{Acq}_{\text{NL}}\}$ for the acquiring of location l such that:

$$\mathsf{so}|_{\text{NLOCK}} = \left\{\langle \mathsf{e}, \mathtt{aRF}_{\mathtt{n}(\mathtt{loc}(\mathsf{e}))}\rangle, \langle \mathsf{e}, \mathtt{aNRW}_{\mathtt{n}(\mathtt{loc}(\mathsf{e}))}\rangle \;\middle|\; \mathtt{m}(\mathsf{e}) = \mathtt{Rel}_{\text{NL}}\right\} \cup \bigcup_{l \in \mathsf{Loc}} \left\{\langle \mathsf{e}_1, \mathtt{aNRW}_{\mathtt{n}(\mathtt{loc}(\mathsf{e}_1))}\rangle, \langle \mathsf{e}_2, \mathtt{aMF}\rangle \;\middle|\; (\mathsf{e}_1, \mathsf{e}_2) \in (\mathsf{po}_l|_{\text{imm}})^{-1}; \mathsf{lo}_l\right\}$$

We define g to map to the first subevent of the implementation. For an event $\mathsf{e}' \in E'$, we choose $g(\mathsf{e}', \mathtt{aMF}) = (\mathsf{e}, \mathtt{aMF})$ with $\mathsf{e} = (t, \_, (\mathtt{Acq}_{\text{NL}}, \_, \_)) \in f^{-1}(\mathsf{e}')$ the first event of the implementation. This g clearly preserves sto (first property of local soundness), as it maps subevents to subevents using the same stamp.

Now we need to pick a suitable so' such that $g(\mathsf{so}') \subseteq \mathsf{hb}$ and $\langle E', \mathsf{po}', \mathtt{stmp}', \mathsf{so}', \_\rangle$ is $\text{RDMA}^{\text{SC}}_{\text{RMW}}$-consistent. I.e., we need well-formed $\mathtt{v}'_{\mathtt{R}}$, $\mathtt{v}'_{\mathtt{W}}$, rf', and mo' such that $g(\mathsf{po}')$, $g(\mathsf{rf}')$, $g(\mathsf{mo}')$, and $g(\mathsf{rb}')$ are all included in hb. We immediately have $g(\mathsf{po}') \in \mathsf{ppo} \subseteq \mathsf{hb}$ since $(\mathtt{aMF}, \mathtt{aMF}) \in \mathsf{sto}$. For the other relations, we can consider each location x independently. Let us note $n = \mathtt{n}(x) = \mathtt{n}(l_x)$. All the relevant operations acquire the lock l_x, as such we can use lo_{l_x} to order them.

We define mo'_x and rf'_x as follows:

$$\mathsf{mo}'_x \triangleq \{(\mathsf{s}'_1, \mathsf{s}'_2) \mid \mathsf{s}'_1, \mathsf{s}'_2 \in \mathcal{G}'.\mathcal{W} \;\wedge\; (g(\mathsf{s}'_1), g(\mathsf{s}'_2)) \in \mathsf{lo}_{l_x}\}$$

$$\mathsf{rf}'_x \triangleq \left\{(\mathsf{s}'_1, \mathsf{s}'_2) \;\middle|\; \begin{array}{c} \mathsf{s}'_1 \in \mathcal{G}'.\mathcal{W} \;\wedge\; \mathsf{s}'_2 \in \mathcal{G}'.\mathcal{R} \;\wedge\; (g(\mathsf{s}'_1), g(\mathsf{s}'_2)) \in \mathsf{lo}_{l_x} \;\wedge \\ \forall \mathsf{s}'_0.(\mathsf{s}'_1, \mathsf{s}'_0) \in \mathsf{mo}'_x \implies (g(\mathsf{s}'_0), g(\mathsf{s}'_2)) \notin \mathsf{lo}_{l_x} \end{array}\right\}$$

with the slight abuse of notation of writing $((\mathsf{e}_1, a_1), (\mathsf{e}_2, a_2)) \in \mathsf{lo}_l$ to mean $(\mathsf{e}_1, \mathsf{e}_2) \in \mathsf{lo}_l$. I.e., the location x is modified in the order of the acquires, and reads read from the latest previous write.

We define $\mathtt{v}'_{\mathtt{R}}$ and $\mathtt{v}'_{\mathtt{W}}$ from the values of $\mathtt{v_R}$ and $\mathtt{v_W}$ on the RDMA subevent (on x) of the implementation. E.g., for e' running $\mathtt{FAA}_{\text{SC}}(x, v)$, there is an event $\mathsf{e} = (\_, \_, (\mathtt{RFAA}, (\_, x, v, \_), ()))$ $\in f^{-1}(\mathsf{e}')$ and we note $\mathtt{v}'_{\mathtt{R}}((\mathsf{e}', \mathtt{aMF})) = \mathtt{v_R}((\mathsf{e}, \mathtt{aNAR}_n))$ and $\mathtt{v}'_{\mathtt{W}}((\mathsf{e}', \mathtt{aMF})) = \mathtt{v_W}((\mathsf{e}, \mathtt{aNRW}_n))$.

We can easily see that $g(\mathsf{rf}')$, $g(\mathsf{mo}')$, and $g(\mathsf{rb}')$ are all included in hb by design. This comes from the fact that $(\mathsf{e}_1, \mathsf{e}_2) \in \mathsf{lo}_{l_x}$ implies $((\mathsf{e}_1, \mathtt{aMF}), (\mathsf{e}_2, \mathtt{aMF})) \in \mathsf{ppo}; \mathsf{so} \subseteq \mathsf{hb}$ (since E respects nodes and the release operation exists) and for rb' because lo_{l_x} is total on the acquiring of the lock l_x (Thus if $(g(\mathsf{s}'_0), g(\mathsf{s}'_2)) \notin \mathsf{lo}_{l_x}$ and $\mathsf{s}'_0 \neq \mathsf{s}'_2$ then $(g(\mathsf{s}'_2), g(\mathsf{s}'_0)) \in \mathsf{lo}_{l_x}$).

The remaining part of the proof is to show that $\mathtt{v}'_{\mathtt{R}}$, $\mathtt{v}'_{\mathtt{W}}$, and rf' are well-formed.

Firstly, let's consider $\mathtt{v}'_{\mathtt{W}}$. For RMW operations, the value is correct from the well-formedness of $\mathtt{v}_{\mathtt{W}}$. For an event $\mathtt{e}'$ running $\mathtt{Write}_{\mathrm{SC}}(x, v)$, the implementation contains $\mathtt{Write}(p^t_x, v); \mathtt{Put}(x, p^t_x, \_)$ (let's call them $\mathtt{e}_1$ and $\mathtt{e}_2$), and we need to show $\mathtt{v}'_{\mathtt{W}}((\mathtt{e}', \mathtt{aMF})) = v$. By definition $\mathtt{v}'_{\mathtt{W}}((\mathtt{e}', \mathtt{aMF})) = \mathtt{v}_{\mathtt{W}}((\mathtt{e}_2, \mathtt{aNRW}_n)) = \mathtt{v}_{\mathtt{R}}((\mathtt{e}_2, \mathtt{aNLR}_n))$ and $\mathtt{v}_{\mathtt{W}}((\mathtt{e}_1, \mathtt{aCW})) = v$. To conclude, it is enough to show $((\mathtt{e}_1, \mathtt{aCW}), (\mathtt{e}_2, \mathtt{aNLR}_n)) \in \mathsf{rf}$. Clearly $\mathtt{e}_1$ is finished when we run $\mathtt{e}_2$ (i.e., $((\mathtt{e}_2, \mathtt{aNLR}_n), (\mathtt{e}_1, \mathtt{aCW})) \in \mathsf{rb}$ would create an hb cycle). It is less obvious that $\mathtt{e}_2$ cannot read from a later $\mathtt{Write}(p^t_x, v')$ (let's call it event $\mathtt{e}_3$) of a later operation $\mathtt{Write}_{\mathrm{SC}}(x, v')$ by the same thread. This is because this later operation would need to acquire the lock l_x. By the semantics of NLOCK, this creates a synchronisation (since $\mathtt{e}_1$ is also towards node n), and we have $((\mathtt{e}_2, \mathtt{aNLR}_n), (\mathtt{e}_3, \mathtt{aCW})) \in \mathsf{ppo}; \mathsf{so}|_{\mathrm{NLOCK}}; \mathsf{ppo} \subseteq \mathsf{hb}$. As such, reading from $\mathtt{e}_3$ would create an hb cycle and is not possible.

Secondly, for $\mathtt{v}'_{\mathtt{R}}$ and non-$\mathtt{Write}_{\mathrm{SC}}$ operations, we need to show that the value returned (i.e. by $\mathtt{e}_r$ running $\mathtt{Read}(r_t)$) is the value read by the RDMA operation $\mathtt{e}$ running $m(r_t, x, \ldots, d)$ with $m \in \{\mathtt{Read}_{\mathrm{SC}}, \mathtt{CAS}_{\mathrm{SC}}, \mathtt{FAA}_{\mathrm{SC}}\}$. For this, we simply show $((\mathtt{e}, \mathtt{aNLW}_n), (\mathtt{e}_r, \mathtt{aCR})) \in \mathsf{rf}$. From the in-between $\mathtt{Wait}$ operation, we have $((\mathtt{e}, \mathtt{aNLW}_n), (\mathtt{e}_r, \mathtt{aCR})) \in \mathsf{pfg}; \mathsf{ppo} \subseteq \mathsf{hb}$. Thus, $\mathtt{e}_r$ cannot ignore $\mathtt{e}$ (i.e. rb would create an hb cycle), and cannot read from a later operations (it would create an ib cycle).

Finally, we are left with checking that rf' is well-formed. We need to show that whenever $(\mathtt{s}'_1, \mathtt{s}'_2) \in \mathsf{rf}'$, with $\mathtt{s}'_i = (\mathtt{e}'_i, \mathtt{aMF})$, we have $\mathtt{v}'_{\mathtt{W}}(\mathtt{s}'_1) = \mathtt{v}'_{\mathtt{R}}(\mathtt{s}'_2)$. (Technically, also that $(\_, \mathtt{s}'_2) \notin \mathsf{rf}'$ implies $\mathtt{v}'_{\mathtt{R}}(\mathtt{s}'_2) = 0$, but this follows from a similar reasoning.) Let $\mathtt{e}_i$ be the RDMA operation in the implementation of $\mathtt{e}'_i$, by definition we have $\mathtt{v}'_{\mathtt{W}}(\mathtt{s}'_1) = \mathtt{v}_{\mathtt{W}}((\mathtt{e}_1, \mathtt{aNRW}_n))$ and $\mathtt{v}'_{\mathtt{R}}(\mathtt{s}'_2) = \mathtt{v}_{\mathtt{R}}((\mathtt{e}_2, a_2))$ (with $a_2 \in \{\mathtt{aNAR}_n, \mathtt{aNRR}_n\}$ depending on the case). Our sufficient goal is then to show that we necessarily have $((\mathtt{e}_1, \mathtt{aNRW}_n), (\mathtt{e}_2, a_2)) \in \mathsf{rf}$. By definition of rf', we have $(g(\mathtt{s}'_1), g(\mathtt{s}'_2)) \in \mathsf{lo}_{l_x}$ as well as $\forall \mathtt{s}'_0 \in \mathcal{G}'.\mathcal{W}.\ ((\mathtt{s}'_1, \mathtt{s}'_0) \in \mathsf{mo}'_x \implies (g(\mathtt{s}'_0), g(\mathtt{s}'_2)) \notin \mathsf{lo}_{l_x})$.

The first point implies $((\mathtt{e}_1, \mathtt{aNRW}_n), (\mathtt{e}_2, a_2)) \in \mathsf{ppo}; \mathsf{so}|_{\mathrm{NLOCK}}; \mathsf{ppo} \subseteq \mathsf{hb}$ by the semantics of locks. This makes $((\mathtt{e}_2, a_2), (\mathtt{e}_1, \mathtt{aNRW}_n)) \in \mathsf{rb}$ impossible (hb cycle), and $\mathtt{e}_2$ reads from either $\mathtt{e}_1$ or a later write: there exists an RDMA operation $\mathtt{e}_3$ (in the implementation of some $\mathtt{e}'_3$) such that $((\mathtt{e}_3, \mathtt{aNRW}_n), (\mathtt{e}_2, a_2)) \in \mathsf{rf}$ with $(\mathtt{e}_1, \mathtt{aNRW}_n) \xrightarrow{\mathsf{mo}^*} (\mathtt{e}_3, \mathtt{aNRW}_n)$. Note that $\mathtt{e}_2 \neq \mathtt{e}_3$ or it would create a $\mathsf{rf}_{\mathtt{e}}; \mathsf{iso}$ cycle in hb; i.e. an event cannot read from itself. Thus we need to show $\mathtt{e}_1 = \mathtt{e}_3$, and by contradiction let us assume $(\mathtt{e}_1, \mathtt{aNRW}_n) \xrightarrow{\mathsf{mo}} (\mathtt{e}_3, \mathtt{aNRW}_n)$.

We then show $(\mathtt{e}'_1, \mathtt{aMF}) \xrightarrow{\mathsf{mo}'} (\mathtt{e}'_3, \mathtt{aMF})$. Since lo_{l_x} is a total order, we have either $(g(\mathtt{s}'_1), g(\mathtt{s}'_3)) \in \mathsf{lo}_{l_x}$ or $(g(\mathtt{s}'_3), g(\mathtt{s}'_1)) \in \mathsf{lo}_{l_x}$. To show the first, we assume the second by contradiction, i.e. that $\mathtt{e}'_3$ acquires first. Given the implementation $I_{\mathtt{SC}}$, there is a $\mathtt{Rel}_{\mathrm{NL}}(x)$ event $\mathtt{e}^r_3$ such that $g(\mathtt{e}'_3) \xrightarrow{\mathsf{po}} \mathtt{e}_3 \xrightarrow{\mathsf{po}} \mathtt{e}^r_3$. Thus from the semantics of NLOCK we have an hb cycle $(\mathtt{e}_3, \mathtt{aNRW}_n) \xrightarrow{\mathsf{ppo}} (\mathtt{e}^r_3, \mathtt{aNRW}_n) \xrightarrow{\mathsf{so}|_{\mathrm{NLOCK}}} g(\mathtt{e}'_1) \xrightarrow{\mathsf{ppo}} (\mathtt{e}_1, \mathtt{aNRW}_n) \xrightarrow{\mathsf{mo}} (\mathtt{e}_3, \mathtt{aNRW}_n)$ providing a contradiction, and we necessarily have $(\mathtt{e}'_1, \mathtt{aMF}) \xrightarrow{\mathsf{mo}'} (\mathtt{e}'_3, \mathtt{aMF})$.

Now, from the definition of rf', the fact lo_{l_x} is a total order, and that $\mathtt{e}_3 \neq \mathtt{e}_2$, we have $(g(\mathtt{s}'_2), g(\mathtt{s}'_3)) \in \mathsf{lo}_{l_x}$. Similarly to previously, this implies $((\mathtt{e}_2, a_2), (\mathtt{e}_3, \mathtt{aNRW}_n)) \in \mathsf{ppo}; \mathsf{ppo}; \mathsf{so}|_{\mathrm{NLOCK}}; \mathsf{ppo} \subseteq \mathsf{hb}$ by the semantics of locks (using both the $\mathtt{aRF}_n$ and

$\texttt{aNLW}_n$ stamps of the release). This contradicts $((\mathsf{e}_3, \texttt{aNRW}_n), (\mathsf{e}_2, a_2)) \in \mathsf{rf}_\mathsf{e} \subseteq \mathsf{hb}$. Thus $\mathsf{e}_1 = \mathsf{e}_3$, $((\mathsf{e}_1, \texttt{aNRW}_n), (\mathsf{e}_2, a_2)) \in \mathsf{rf}$, and rf' is well-formed.

C Declarative Semantics of $\text{RDMA}^{\text{TSO}}_{\text{RMW}}$ à la MOWGLI

In this appendix, we first (§C.1) present the declarative semantics of $\text{RDMA}^{\text{TSO}}_{\text{RMW}}$ in a format similar to that of RDMA^{TSO} in [4], but extended with remote RMW operations similarly to the semantics of $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ given in §3. It is slightly different from the one in §D, as we use the stamps and subevents system of MOWGLI.

We then (§C.2) provide a definition of the implementation of $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ into $\text{RDMA}^{\text{TSO}}_{\text{RMW}}$. Finally (§C.3), we give a proof of the soundness of this implementation, similarly to [4].

C.1 Semantics

Our definition of $\text{RDMA}^{\text{TSO}}_{\text{RMW}}$ is closer to an independent language than a library. We do not need a relation hb to represent the potential rest of the program, as a program *cannot* combine instructions from $\text{RDMA}^{\text{TSO}}_{\text{RMW}}$ and other libraries presented in this paper.

We use the following 13 methods:

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

- $\texttt{Write}^{\text{TSO}} : \mathsf{Loc} \times \mathsf{Val} \to ()$
- $\texttt{Read}^{\text{TSO}} : \mathsf{Loc} \to \mathsf{Val}$
- $\texttt{CAS}^{\text{TSO}} : \mathsf{Loc} \times \mathsf{Val} \times \mathsf{Val} \to \mathsf{Val}$
- $\texttt{Mfence}^{\text{TSO}} : () \to ()$
- $\texttt{Get}^{\text{TSO}} : \mathsf{Loc} \times \mathsf{Loc} \to \mathsf{Val}$
- $\texttt{Put}^{\text{TSO}} : \mathsf{Loc} \times \mathsf{Loc} \to \mathsf{Val}$
- $\texttt{Poll} : \mathsf{Node} \to \mathsf{Val}$
- $\texttt{Rfence}^{\text{TSO}} : \mathsf{Node} \to ()$
- $\texttt{RCAS}^{\text{TSO}} : \mathsf{Loc} \times \mathsf{Loc} \times \mathsf{Val}^2 \to \mathsf{Val}$
- $\texttt{RFAA}^{\text{TSO}} : \mathsf{Loc} \times \mathsf{Loc} \times \mathsf{Val} \to \mathsf{Val}$
- $\texttt{SetAdd} : \mathsf{Loc} \times \mathsf{Val} \to ()$
- $\texttt{SetRemove} : \mathsf{Loc} \times \mathsf{Val} \to ()$
- $\texttt{SetIsEmpty} : \mathsf{Loc} \to \mathbb{B}$

This version is based on [4] extended with remote RMW. Compared to RDMA^{TSO} from [3], we slightly extend the language so that RDMA operations return an arbitrary unique identifier, and polling also returns the same identifier of the operation being polled. In addition, we also assume basic set operations `SetAdd`, `SetRemove`, and `SetIsEmpty` to store these new identifiers, where the locations used for sets do not overlap with locations used for other operations.

Consistency predicate. An execution of an $\text{RDMA}^{\text{TSO}}_{\text{RMW}}$ program is of the form $\mathcal{G} = \langle E, \mathsf{po}, \texttt{stmp}, \mathsf{so} \rangle$. Note that $\mathsf{hb} = (\mathsf{ppo} \cup \mathsf{so})^+$ does not have the flexibility of containing additional external constraints.

We say that a stamping function $\texttt{stmp}_{\texttt{TSO}}$ is valid if:

- Polls have stamp $\texttt{aWT}$: $\texttt{stmp}_{\texttt{TSO}}((\_,\_,(\texttt{Poll},\_,\_))) = \{\texttt{aWT}\}$.
- Auxiliary set operations have stamp $\texttt{aMF}$: $\texttt{stmp}_{\texttt{TSO}}((\_,\_,(\texttt{SetAdd},\_,\_))) = \texttt{stmp}_{\texttt{TSO}}((\_,\_,(\texttt{SetRemove},\_,\_))) = \texttt{stmp}_{\texttt{TSO}}((\_,\_,(\texttt{SetIsEmpty},\_,\_))) = \{\texttt{aMF}\}$.
- Other events follow the validity constraints of $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ (*cf.* Section 3). E.g., events calling $\texttt{Write}^{\text{TSO}}$ have stamp $\texttt{aCW}$, while events calling $\texttt{Get}^{\text{TSO}}$ towards node n have stamps $\texttt{aNRR}_n$ and $\texttt{aNLW}_n$. We also define $\texttt{loc}$ on subevents similarly to $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$.

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

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

- $\texttt{v}_\texttt{R}$, $\texttt{v}_\texttt{W}$, rf, mo, nfo, and rao are well-formed, as in $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$;
- Let $P_n \triangleq \{(\mathsf{e}, \texttt{aWT}) \mid \mathsf{e} = (\_,\_,(\texttt{Poll},(n),\_)) \in E\}$ be the set of poll (sub)events towards node n. Let $C_n \triangleq \left\{(\mathsf{e}, \texttt{aNLW}_n) \;\middle|\; \texttt{m}(\mathsf{e}) \in \left\{\texttt{Get}^{\text{TSO}}, \texttt{RCAS}^{\text{TSO}}, \texttt{RFAA}^{\text{TSO}}\right\}\right\} \cup \left\{(\mathsf{e}, \texttt{aNRW}_n) \;\middle|\; \texttt{m}(\mathsf{e}) = \texttt{Put}^{\text{TSO}}\right\}$ be the set of (final writes of) remote operations towards node n that need polling. Note that, for remote RMW operations, polling only synchronises with the $\texttt{aNLW}_n$ part and not with the (potential) $\texttt{aNRW}_n$ part.
  Then $\mathsf{pf} \subseteq \bigcup_{n \in \mathsf{Node}} C_n \times P_n$ is the *polls-from* relation, relating earlier NIC writes to later polls. Moreover:
  - $\mathsf{pf} \subseteq \mathsf{po}$ (we can only poll previous operations of the same thread);
  - pf is functional on its domain (every NIC write can be polled at most once);
  - pf is total and functional on its range (every $\texttt{Poll}$ polls from exactly one NIC write);
  - $\texttt{Poll}$ events poll-from the oldest non-polled remote operation towards the given node:
    for each node n, if $w_1, w_2 \in C_n$ and $w_1 \xrightarrow{\mathsf{po}} w_2 \xrightarrow{\mathsf{pf}} p_2$, then there exists p_1 such that $w_1 \xrightarrow{\mathsf{pf}} p_1 \xrightarrow{\mathsf{po}} p_2$;
  - and a $\texttt{Poll}$ returns the unique identifier of the polled operation:
    if $((\_,\_,(\_,\_,v_1)),\_) \xrightarrow{\mathsf{pf}} ((\_,\_,(\texttt{Poll},\_,v_2)),\texttt{aWT})$ then $v_1 = v_2$.

We use the derived relations rb, $\mathsf{rb_i}$, $\mathsf{rf_e}$, $\mathsf{rf_i}$, ippo, and iso as defined for $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$. We can then define ib as follows:

$$\mathsf{ib} \triangleq (\mathsf{ippo} \cup \mathsf{iso} \cup \mathsf{rf} \cup \mathsf{pf} \cup \mathsf{nfo} \cup \mathsf{rb_i} \cup \{(\mathsf{e}, \texttt{aNRW}_n), (\mathsf{e}, \texttt{aNLW}_n)\})^+$$

The last new component states that, for remote RMW operations, the remote write part *starts* before the local write part. As mentioned previously, this does not imply that they finish in order, and this component is not included in so. Since it does not prevent any behaviour, we remove this component in the definition $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ (§3), but we keep it here as it simplifies the soundness proof (Theorem 6) and the equivalence proof with the operational semantics (Appendix D.5).

Definition 16 ($\text{RDMA}^{\text{TSO}}_{\text{RMW}}$-consistency). *$\mathcal{G} = \langle E, \mathsf{po}, \mathtt{stmp}, \mathsf{so}\rangle$ is $\text{RDMA}^{\text{TSO}}_{\text{RMW}}$-consistent if:*

- $(\mathsf{ppo} \cup \mathsf{so})^+$ *is irreflexive;*
- $\langle E, \mathsf{po}\rangle$ *respects nodes (as in* $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$*);*
- $\mathtt{stmp}$ *is valid;*
- *there exists well-formed* $\mathtt{v_R}$, $\mathtt{v_W}$, rf, mo, nfo, pf, *and* rao *such that* ib *is irreflexive and* $\mathsf{so} = \mathsf{iso} \cup \mathsf{rf_e} \cup [\mathtt{aNLW}]; \mathsf{pf} \cup \mathsf{nfo} \cup \mathsf{rb} \cup \mathsf{mo} \cup \mathsf{rao} \cup ([\mathtt{aNRW}]; \mathsf{iso}^{-1}; \mathsf{rao}) \cup ([\mathtt{Inst}]; \mathsf{ib})$*;*
- *identifiers for RDMA operations are unique: if* $\mathtt{e}_1$ *and* $\mathtt{e}_2$ *are both of the form* $(\_, \_, (m, \_, v))$ *with* $m \in \{\mathtt{Put}^{\text{TSO}}, \mathtt{Get}^{\text{TSO}}, \mathtt{RCAS}^{\text{TSO}}, \mathtt{RFAA}^{\text{TSO}}\}$ *then* $\mathtt{e}_1 = \mathtt{e}_2$*;*
- *and the set operations are (per-thread) sound: if* $\mathtt{SetIsEmpty}$ *returns* $\mathtt{true}$*, then every value added to the set was subsequently removed. I.e., if* $\mathtt{e}_1 = (t, \_, (\mathtt{SetAdd}, (x, v), \_))$, $\mathtt{e}_3 = (t, \_, (\mathtt{SetIsEmpty}, (x), \mathtt{true}))$, *and* $\mathtt{e}_1 \xrightarrow{\mathsf{po}} \mathtt{e}_3$*, then there exists* $\mathtt{e}_2 = (t, \_, (\mathtt{SetRemove}, (x, v), \_))$ *such that* $\mathtt{e}_1 \xrightarrow{\mathsf{po}} \mathtt{e}_2 \xrightarrow{\mathsf{po}} \mathtt{e}_3$.

C.2 Implementation Function

In Fig. 16 we define the implementation $I_{\mathtt{W}}$ from a full program using only the $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ library into a program using only $\text{RDMA}^{\text{TSO}}_{\text{RMW}}$. We assume threads use disjoint work identifiers $d \in \mathsf{Wid}$, otherwise it is straightforward to rename them.

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

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

C.3 Soundness

We do not prove that the implementation above is locally sound as it does not apply for this case. Instead, we assume a full program using only the $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ library and compile it into $\text{RDMA}^{\text{TSO}}_{\text{RMW}}$.

Theorem 6. *Let $\widetilde{\mathsf{p}}$ be a program using only the $\text{RDMA}^{\text{WAIT}}_{\text{RMW}}$ library. Then we have $\mathtt{outcome}_{\text{RDMA}^{\text{TSO}}_{\text{RMW}}}(\llbracket\widetilde{\mathsf{p}}\rrbracket_{I_{\mathtt{W}}}) \subseteq \mathtt{outcome}_{\{\text{RDMA}^{\text{WAIT}}_{\text{RMW}}\}}(\widetilde{\mathsf{p}})$, where:*

$$\mathtt{outcome}_{\{\text{RDMA}^{\text{WAIT}}_{\text{RMW}}\}}(\widetilde{\mathsf{p}}) = \{\widetilde{v} \mid \exists \langle E, \mathsf{po}, \mathtt{stmp}, \mathsf{so}, \mathsf{hb}\rangle \ \{\text{RDMA}^{\text{WAIT}}_{\text{RMW}}\}\text{-consistent}.\ \langle \widetilde{v}, \langle E, \mathsf{po}\rangle\rangle \in \llbracket\widetilde{\mathsf{p}}\rrbracket\}$$
$$\mathtt{outcome}_{\text{RDMA}^{\text{TSO}}_{\text{RMW}}}(\llbracket\widetilde{\mathsf{p}}\rrbracket_{I_{\mathtt{W}}}) = \{\widetilde{v} \mid \exists \langle E, \mathsf{po}, \mathtt{stmp}, \mathsf{so}\rangle \ \text{RDMA}^{\text{TSO}}_{\text{RMW}}\text{-consistent}.\ \langle \widetilde{v}, \langle E, \mathsf{po}\rangle\rangle \in \llbracket\llbracket\widetilde{\mathsf{p}}\rrbracket_{I_{\mathtt{W}}}\rrbracket\}$$

$I_{\mathtt{W}}(t, \mathtt{Write}, (x, v)) \triangleq \mathtt{Write}^{\mathrm{TSO}}(x, v)$

$I_{\mathtt{W}}(t, \mathtt{Read}, (x)) \triangleq \mathtt{Read}^{\mathrm{TSO}}(x)$

$I_{\mathtt{W}}(t, \mathtt{CAS}, (x, v_1, v_2)) \triangleq \mathtt{CAS}^{\mathrm{TSO}}(x, v_1, v_2)$

$I_{\mathtt{W}}(t, \mathtt{Mfence}, ()) \triangleq \mathtt{Mfence}^{\mathrm{TSO}}()$

$I_{\mathtt{W}}(t, \mathtt{Rfence}, (n)) \triangleq \mathtt{Rfence}^{\mathrm{TSO}}(n)$

$I_{\mathtt{W}}(t, \mathtt{Get}, (x, y, d)) \triangleq$

$\mathtt{let}\ v = \mathtt{Get}^{\mathrm{TSO}}(x, y)\ \mathtt{in}\ \mathtt{SetAdd}(d^{\mathtt{n}(y)}, v)$

$I_{\mathtt{W}}(t, \mathtt{Put}, (x, y, d)) \triangleq$

$\mathtt{let}\ v = \mathtt{Put}^{\mathrm{TSO}}(x, y)\ \mathtt{in}\ \mathtt{SetAdd}(d^{\mathtt{n}(x)}, v)$

$I_{\mathtt{W}}(t, \mathtt{Wait}, (d)) \triangleq$

$\mathtt{For}\ n\ \mathtt{in}\ 1, \ldots, N\ \mathtt{do}\ \{$

$\quad \mathtt{While}\ (\mathtt{SetIsEmpty}(d^n) \neq \mathtt{true})\ \mathtt{do}\ \{$

$\quad\quad \mathtt{let}\ v = \mathtt{Poll}(n)\ \mathtt{in}$

$\quad\quad \mathtt{For}\ k\ \mathtt{in}\ 1, \ldots, K\ \mathtt{do}\ \{$

$\quad\quad\quad \mathtt{SetRemove}(d^n_k, v) \quad \} \} \}$

$I_{\mathtt{W}}(t, \mathtt{RCAS}, (x, y, v_1, v_2, d)) \triangleq$

$\mathtt{let}\ v = \mathtt{RCAS}^{\mathrm{TSO}}(x, y, v_1, v_2)\ \mathtt{in}\ \mathtt{SetAdd}(d^{\mathtt{n}(y)}, v)$

$I_{\mathtt{W}}(t, \mathtt{RFAA}, (x, y, v', d)) \triangleq$

$\mathtt{let}\ v = \mathtt{RFAA}^{\mathrm{TSO}}(x, y, v')\ \mathtt{in}\ \mathtt{SetAdd}(d^{\mathtt{n}(y)}, v)$

Fig. 16: Implementation $I_{\mathtt{W}}$ of $\mathrm{RDMA}^{\mathrm{WAIT}}_{\mathrm{RMW}}$ into $\mathrm{RDMA}^{\mathrm{TSO}}_{\mathrm{RMW}}$

Proof. By definition, we are given $\mathcal{G} = \langle E, \mathsf{po}, \mathtt{stmp}, \mathsf{so}\rangle$ $\mathrm{RDMA}^{\mathrm{TSO}}_{\mathrm{RMW}}$-consistent (Definition 16) such that $\langle \widetilde{v}, \langle E, \mathsf{po}\rangle\rangle \in [\![\lfloor \widetilde{\mathsf{p}} \rfloor_{I_{\mathtt{W}}}]\!]$. Among others, it means $\langle E, \mathsf{po}\rangle$ respects nodes and there exists well-formed $\mathtt{v_R}$, $\mathtt{v_W}$, rf, mo, nfo, pf, and rao such that ib is irreflexive, $\mathtt{stmp}$ is valid, $\mathsf{so} = \mathsf{iso} \cup \mathsf{rf_e} \cup [\cup_n\ \mathtt{aNLW}_n]; \mathsf{pf} \cup \mathsf{nfo} \cup \mathsf{rb} \cup \mathsf{mo} \cup \mathsf{rao} \cup ([\mathtt{aNRW}]; \mathsf{iso}^{-1}; \mathsf{rao}) \cup ([\mathtt{Inst}]; \mathsf{ib})$, and $\mathsf{hb} \triangleq (\mathsf{ppo} \cup \mathsf{so})^+$ is irreflexive.

From Lemma 1, since $\widetilde{\mathsf{p}}$ uses only $\mathrm{RDMA}^{\mathrm{WAIT}}_{\mathrm{RMW}}$, there is E', po', f such that $\langle \widetilde{v}, \langle E', \mathsf{po}'\rangle\rangle \in [\![\widetilde{\mathsf{p}}]\!]$ and $\mathtt{abs}^f_{I_{\mathtt{W}}, \mathrm{RDMA}^{\mathrm{WAIT}}_{\mathrm{RMW}}}(\langle E, \mathsf{po}\rangle, \langle E', \mathsf{po}'\rangle)$. Note that this clearly implies $\langle E, \mathsf{po}\rangle$ also respects nodes, as the implementation $I_{\mathtt{W}}$ keeps the same locations. Our objective is to find $\mathtt{stmp}'$, so', and hb' such that $\mathcal{G}' = \langle E', \mathsf{po}', \mathtt{stmp}', \mathsf{so}', \mathsf{hb}'\rangle$ is $\{\mathrm{RDMA}^{\mathrm{WAIT}}_{\mathrm{RMW}}\}$-consistent (Definitions 2 and 8). To choose a valid function $\mathtt{stmp}'$, most values are forced. For remote compare-and-swap, we make the same choice as $\mathtt{stmp}$. I.e. for each $\mathtt{RCAS}$ we assert it succeeds iff the corresponding $\mathtt{RCAS}^{\mathrm{TSO}}$ in its implementation succeeds. We will also pick $\mathsf{hb}' \triangleq (\mathsf{ppo}' \cup \mathsf{so}')^+$ since there is no external constraints. Thus, we only need to carefully pick so' and show it works.

While our objective is not exactly local soundness (Definition 15), we still use a concretisation function $g : \langle E', \mathsf{po}', \mathtt{stmp}'\rangle.\mathsf{SEvent} \to \mathcal{G}.\mathsf{SEvent}$ to then define so'.

- For $\mathsf{e}' = (t, \_, (\mathtt{Write}, (x, v), ()))$, from the definition of the implementation $I_{\mathtt{W}}$ and the abstraction f, there is some event $\mathsf{e} = (t, \_, (\mathtt{Write}^{\mathrm{TSO}}, (x, v), ())) \in f^{-1}(\mathsf{e}')$. We define $g(\mathsf{e}', \mathtt{aCW}) = (\mathsf{e}, \mathtt{aCW})$. For events calling $\mathtt{Read}$, $\mathtt{CAS}$, $\mathtt{Mfence}$, and $\mathtt{Rfence}$, we proceed similarly and let g map each subevent to their counterpart in the implementation.
- For $\mathsf{e}' = (t, \_, (\mathtt{Get}, (x, y, d), ()))$, there is some event $\mathsf{e} = (t, \_, (\mathtt{Get}^{\mathrm{TSO}}, (x, y), (v))) \in f^{-1}(\mathsf{e}')$. We define $g(\mathsf{e}', \mathtt{aNRR}_{n(y)}) = (\mathsf{e}, \mathtt{aNRR}_{n(y)})$ and $g(\mathsf{e}', \mathtt{aNLW}_{n(y)}) = (\mathsf{e}, \mathtt{aNLW}_{n(y)})$. We proceed similarly for $\mathtt{Put}$, $\mathtt{RCAS}$, and $\mathtt{RFAA}$ events.

- Finally for $\mathsf{e}' = (t, \_, (\texttt{Wait}, (d), ()))$, there is in $f^{-1}(\mathsf{e}')$ some last event (in po order) of the form $\mathsf{e} = (t, \_, (\texttt{SetIsEmpty}, (d^N), \texttt{true}))$ confirming the set d^N tracking operations towards the last node N is empty. We define $g(\mathsf{e}', \texttt{aWT}) = (\mathsf{e}, \texttt{aMF})$.

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

Each subevent in $\mathcal{G}'.\mathcal{R}$ (resp. $\mathcal{G}'.\mathcal{W}$) is mapped through g to a subevent in $\mathcal{G}.\mathcal{R}$ (resp. $\mathcal{G}.\mathcal{W}$) using the same stamp and location. Thus it is straightforward to define $\mathsf{v}'_{\mathtt{R}}$, $\mathsf{v}'_{\mathtt{W}}$, rf', mo', nfo', and rao' by relying on their counterparts in $\mathcal{G}$. E.g. $\mathsf{v}'_{\mathtt{R}}(\mathsf{s}') \triangleq \mathsf{v}_{\mathtt{R}}(g(\mathsf{s}'))$ and $\mathsf{rf}' \triangleq \{(\mathsf{s}'_1, \mathsf{s}'_2) \mid (g(\mathsf{s}'_1), g(\mathsf{s}'_2)) \in \mathsf{rf}\}$. The well-formedness of $\mathsf{v}_{\mathtt{R}}$, $\mathsf{v}_{\mathtt{W}}$, rf, mo, nfo, and rao trivially implies that of $\mathsf{v}'_{\mathtt{R}}$, $\mathsf{v}'_{\mathtt{W}}$, rf', mo', nfo', and rao'. From this, we can define all the expected derived relations, including pfg', pfp', and $\mathsf{ib}' \triangleq (\mathsf{ippo}' \cup \mathsf{iso}' \cup \mathsf{rf}' \cup \mathsf{pfg}' \cup \mathsf{pfp}' \cup \mathsf{nfo}' \cup \mathsf{rb}'_{\mathsf{i}})^+$. We then define $\mathsf{so}' \triangleq \mathsf{iso}' \cup \mathsf{rf}'_{\mathsf{e}} \cup \mathsf{pfg}' \cup \mathsf{nfo}' \cup \mathsf{rb}' \cup \mathsf{mo}' \cup \mathsf{rao}' \cup ([\texttt{aNRW}]; \mathsf{iso}'^{-1}; \mathsf{rao}') \cup ([\texttt{Inst}]; \mathsf{ib}')$, and as previously mentioned $\mathsf{hb}' \triangleq (\mathsf{ppo}' \cup \mathsf{so}')^+$.

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

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

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

Let us start with pfg', assuming $((\mathsf{e}'_1, \texttt{aNLW}_n), (\mathsf{e}'_2, \texttt{aWT})) \in \mathsf{pfg}'$. By definition they are of the form $\mathsf{e}'_1 = (t, \_, (\_, (\ldots, d), ()))$ and $\mathsf{e}'_2 = (t, \_, (\texttt{Wait}, (d), ()))$, for some t, d, and $\mathtt{m}(\mathsf{e}'_1) \in \{\texttt{Get}, \texttt{RCAS}, \texttt{RFAA}\}$, with $(\mathsf{e}'_1, \mathsf{e}'_2) \in \mathsf{po}'$ and n the remote node of this operation. By definition of the implementation and the abstraction, $f^{-1}(\mathsf{e}'_1)$ contains two events $\mathsf{e}_1 = (t, \_, (m, (\ldots), (v)))$ with a similar method $m \in \{\texttt{Get}^{\text{TSO}}, \texttt{RCAS}^{\text{TSO}}, \texttt{RFAA}^{\text{TSO}}\}$ and $\mathsf{e}_a = (t, \_, (\texttt{SetAdd}, (d^n, v), ()))$, with $\mathsf{e}_1 \xrightarrow{\mathsf{po}} \mathsf{e}_a$. Meanwhile $f^{-1}(\mathsf{e}'_2)$ contains a last event $\mathsf{e}_2 = (t, \_, (\texttt{SetIsEmpty}, (d^N), \texttt{true}))$ and an earlier event $\mathsf{e}_3 = (t, \_, (\texttt{SetIsEmpty}, (d^n), \texttt{true}))$, with $\mathsf{e}_3 \xrightarrow{\mathsf{po}^*} \mathsf{e}_2$, confirming operations towards n are done (if $n = N$ then $\mathsf{e}_2 = \mathsf{e}_3$).

Since $f(\mathsf{e}_a) = \mathsf{e}'_1 \xrightarrow{\mathsf{po}'} \mathsf{e}'_2 = f(\mathsf{e}_3)$ and f is an abstraction, we have $\mathsf{e}_a \xrightarrow{\mathsf{po}} \mathsf{e}_3$, i.e. the value v is added to d^n before the moment d^n is confirmed empty. By consistency (Definition 16), there is an in-between event $\mathsf{e}_4 = (t, \_, (\texttt{SetRemove}, (d^n, v), ()))$ that removes this value, with $\mathsf{e}_a \xrightarrow{\mathsf{po}} \mathsf{e}_4 \xrightarrow{\mathsf{po}} \mathsf{e}_3$. From the definition of the implementation, such an event e_4 is immediately preceded (with maybe other $\texttt{SetRemove}$ in-between) by an event $\mathsf{e}_p = (t, \_, (\texttt{Poll}, (n), (v)))$. Now we argue that we necessarily have $((\mathsf{e}_1, \texttt{aNLW}_n), (\mathsf{e}_p, \texttt{aWT})) \in \mathsf{pf}$. From the well-formedness of pf, we know that $(\mathsf{e}_p, \texttt{aWT})$ has a preimage (pf is total and functional on its

range) and that this preimage outputs the value v. By consistency (Definition 16), e_1, with $\mathtt{m}(\mathsf{e}_1) \in \{\mathtt{Get}^{\mathrm{TSO}}, \mathtt{RCAS}^{\mathrm{TSO}}, \mathtt{RFAA}^{\mathrm{TSO}}\}$, is the only RDMA operation with output v. Thus $(\mathsf{e}_1, \mathtt{aNLW}_n)$ is the preimage of $(\mathsf{e}_p, \mathtt{aWT})$ by pf.

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

For pfp' we have two cases. First, for a $\mathtt{Put}$ operation e'_1, having $((\mathsf{e}'_1, \mathtt{aNRW}_n), (\mathsf{e}'_2, \mathtt{aWT})) \in \mathsf{pfp}'$ similarly implies $g(\mathsf{e}'_1, \mathtt{aNRW}_n) \xrightarrow{\mathsf{pf};\mathsf{ppo}^+} g(\mathsf{e}'_2, \mathtt{aWT})$ for the same reasons. Second, for a successful remote RMW operation e'_1, having $((\mathsf{e}'_1, \mathtt{aNRW}_n), (\mathsf{e}'_2, \mathtt{aWT})) \in \mathsf{pfp}'$, with $g(\mathsf{e}'_1, \mathtt{aNRW}_n) = (\mathsf{e}_1, \mathtt{aNRW}_n)$ instead implies $(\mathsf{e}_1, \mathtt{aNLW}_n) \xrightarrow{\mathsf{pf};\mathsf{ppo}^+} g(\mathsf{e}'_2, \mathtt{aWT})$ for the same reasons. This is because the semantics of polls synchronises with the *local write* part of the operation, not with the remote write part. However, we do have $g(\mathsf{e}'_1, \mathtt{aNRW}_n) \xrightarrow{\mathsf{ib}} (\mathsf{e}_1, \mathtt{aNLW}_n)$ from the last component of ib, and thus $g(\mathsf{pfg}') \subseteq \mathsf{ib}^?; \mathsf{pf}; \mathsf{ppo}^+$.

Thus ib' and hb' are irreflexive, and $\mathcal{G}'$ is $\{\mathrm{RDMA}^{\mathrm{WAIT}}_{\mathrm{RMW}}\}$-consistent.

D The $\mathrm{RDMA}^{\mathrm{TSO}}_{\mathrm{RMW}}$ Memory Model

In this section, we present an operational (§D.1) and declarative model (§D.2) for $\mathrm{RDMA}^{\mathrm{TSO}}_{\mathrm{RMW}}$ in the format of $\mathrm{RDMA}^{\mathrm{TSO}}$ as defined in [3], as well as an extension of their equivalence proof (§D.3 onwards).

The declarative format used in §D.2 (based on [3]) is slightly different from the one of C.1 (based on [4]), but they represent the same semantics.

D.1 Operational Semantics

Nodes and Threads. We write $\mathsf{Node} = \{1..N\}$ for the set of node identifiers, and Tid for the set of thread identifiers. We write n (resp. t) to range over nodes (resp. threads), and given some node n we write $\overline{n}$ to range over the set of all other nodes $\mathsf{Node} \setminus \{n\}$. Each thread runs on a particular node, so we write $n(t)$ for the node the thread belongs to.

Note that the semantics of [3] assumes that the remote node $\overline{n}$ of an operation is different from the local node n (i.e. they ignore loopback). As we extend their operational model, we keep their notations. However, as shown in [4], loopbacks are possible and follow exactly the same semantics.

Memory. Although all nodes can directly access all memory locations, whether an operation is towards local or remote memory is pivotal to our semantics, so we are always careful to note the node to which a memory location belongs. We write Loc_n for the set of locations local to node n, and $\mathsf{Loc} = \biguplus_n \mathsf{Loc}_n$ for the set of all locations. We use $\mathsf{Loc}_{\overline{n}} = \mathsf{Loc} \setminus \mathsf{Loc}_n$ and write x^n, y^n, z^n for values in Loc_n, respectively $x^{\overline{n}}, y^{\overline{n}}, z^{\overline{n}}$ for $\mathsf{Loc}_{\overline{n}}$. When the node in question is sufficiently clear, we elide the superscript and instead simply write x or $\overline{x}$ for local or remote locations respectively.

Values and Expressions. The language of expressions is standard and elided. We write $v \in \mathsf{Val}$ for values, with $\mathbb{N} \subseteq \mathsf{Val}$, and $e \in \mathsf{Exp}$ for expressions. We write $\mathsf{elocs}(e)$ for the set of memory locations referenced in e, $e[v/x]$ for the expression obtained by substituting all references to location x in e with value v, and $[\![e]\!]$ for the evaluation of e given it is *closed*, that is, $\mathsf{elocs}(e) = \emptyset$. We use e^n for expressions where $\mathsf{elocs}(e^n) \subseteq \mathsf{Loc}_n$

Commands and Programs. Commands are described by the C^n grammar below. CPU operations (CComm) are assignment, assumption of the value of a location, memory fence, compare-and-swap, and poll, which awaits the earliest completion notification of a remote operations towards $\overline{n}$.

RDMA operations (RComm) are either a 'get' of the form $x := \overline{y}$ which reads a remote location $\overline{y}$ and writes its value to local location x, a 'put' ($\overline{y} := x$) which does the reverse, 'remote-CAS' (resp. 'remote-FAA') which executes a *remote* compare-and-swap (resp. fetch-and-add), and 'remote fence' which ensures all prior RDMA operations towards $\overline{n}$ complete before any later RDMA operations towards $\overline{n}$ execute. We note rRMW to cover both kind of remote read-modify-write operations, i.e. `RCAS` and `RFAA`.

Primitive operations (PComm) are CPU or RDMA operations, and commands (Comm) are the no-op, primitive operations, sequential composition (executes the first command, then the second), non-deterministic choice (executes one command or the other), and non-deterministic loop (executes the command some finite, possibly zero number of times).

A program P consists of a map from threads to commands, such that each $t \in \mathsf{Tid}$ is mapped to a command on $n(t)$.

$$\mathsf{Comm} \ni \mathsf{C}^n ::= \mathtt{skip} \mid \mathsf{c}^n \mid \mathsf{C}_1^n; \mathsf{C}_2^n \mid \mathsf{C}_1^n + \mathsf{C}_2^n \mid \mathsf{C}^{n*} \qquad \mathsf{PComm} \ni \mathsf{c}^n ::= \mathsf{cc}^n \mid \mathsf{rc}^n$$
$$\mathsf{CComm} \ni \mathsf{cc}^n ::= x := e^n \mid \mathtt{assume}(x = v) \mid \mathtt{assume}(x \neq v) \mid \mathtt{mfence} \mid x := \mathtt{CAS}(y, e_1, e_2) \mid \mathtt{poll}(\overline{n})$$
$$\mathsf{RComm} \ni \mathsf{rc}^n ::= x := \overline{y} \mid \overline{y} := x \mid x := \mathtt{RCAS}(\overline{y}, e_1, e_2) \mid x := \mathtt{RFAA}(\overline{y}, e) \mid \mathtt{rfence}(\overline{n})$$

Store Buffers. To permit the weak behaviours of TSO (i.e. write-read reordering), we assign each thread a *store buffer* $\mathsf{B}(t)$, which is a FIFO queue containing pending writes to memory by that thread. When a thread performs a CPU read, it reads the most recent entry for that location in its store buffer, if there is one, instead of the value in memory. The write at the head of the queue may be flushed to memory at any time, and `mfence` and `CAS` wait until the store buffer is empty before executing.

Queue Pairs. We follow the *simplified* operational model described in [3], and therefore consider a queue-pair structure comprising three FIFO queues: **pipe**, which contains pending or in-progress RDMA operations; $\mathbf{wb_R}$, the remote write buffer, which contains pending writes to the memory of the remote node; and $\mathbf{wb_L}$, the local write buffer, which contains pending writes to the memory of the local node. The structure is shown in Fig. 17. Notice that under this simplified model, the transition between local and remote node in **pipe** is continuous – we do not explicitly model the transition between local (yellow) and remote (pink) sides.

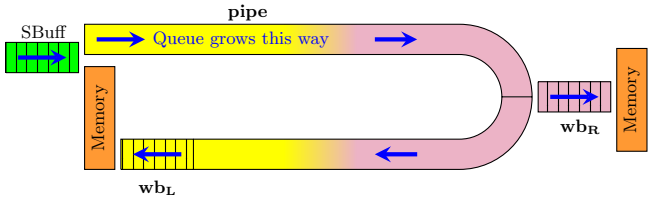


Fig. 17: Simple queue-pair structure.

Remote Atomics. To model the behaviour of RDMA atomic operations, we assign each node a *remote atomic lock* $\mathsf{A}(n)$, which is a boolean indicating whether an RDMA atomic is currently in progress *towards* that node.

Transitions of the Operational Semantics. We describe the rules governing the transitions between states, which comprise a program P, global memory M, store buffers B, queue pairs QP and remote atomic locks A.

Transitions take the form shown on the right, which should be read as: if ϕ is true, then it is allowed for the system to transition from the state described by $\mathsf{P} \ldots \mathsf{QP}$ to the state described by $\mathsf{P}' \ldots \mathsf{QP}'$.

$$\frac{\phi}{\mathsf{P}, \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP} \Rightarrow \mathsf{P}', \mathsf{M}', \mathsf{B}', \mathsf{A}', \mathsf{QP}'}$$

In practice, however, writing each transition rule in such a way would be verbose and hard to understand, as most transitions do not affect every part of the state. We can separate *program transitions* concerning P from *hardware transitions* concerning $\mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}$. In order to synchronise the two where necessary, we assign labels to certain transitions and require that a labelled program transition only occur if it is matched by a hardware transition with the same label (or vice-versa). Labels are of the form $t : l$ where t is the thread executing at that step and $l \in \mathsf{Lab}$ is the label of the operation. *Silent transition*, which affect only the program (resp. only hardware) are written with the empty label, ϵ, and may be taken independently.

Fig. 18 shows the top-level rules of the operational semantics which govern this separation. We can henceforth consider the program and hardware transitions separately.

Program Transitions. Fig. 19 shows the program and command transitions (middle), labels (above) and expression rewriting rules (below). The transitions for non-remote commands are familiar from TSO. Notice that the transitions for get and put simply transition to `skip` with the relevant label; we know that this means there will be some relevant transition in the hardware. The transition to `skip` allows the program to continue executing, which we expect, as remote operations are handled asynchronously by the NIC.

This is similarly the case for the rules for remote-CAS and remote-FAA. The expressions involved are required to be closed, similarly to the rules for local write and CAS; the expressions must be evaluated before the transition.

$$\frac{\mathsf{P} \xrightarrow{t:\varepsilon} \mathsf{P}'}{\mathsf{P},\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \Rightarrow \mathsf{P}',\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}} \qquad \frac{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{t:\varepsilon} \mathsf{M}',\mathsf{B}',\mathsf{A}',\mathsf{QP}'}{\mathsf{P},\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \Rightarrow \mathsf{P},\mathsf{M}',\mathsf{B}',\mathsf{A}',\mathsf{QP}'}$$

$$\frac{\mathsf{P} \xrightarrow{t:l} \mathsf{P}' \quad \mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{t:l} \mathsf{M}',\mathsf{B}',\mathsf{A}',\mathsf{QP}'}{\mathsf{P},\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \Rightarrow \mathsf{P}',\mathsf{M}',\mathsf{B}',\mathsf{A}',\mathsf{QP}'}$$

Fig. 18: RDMA$^{\text{TSO}}$ operational semantics with the program and hardware transitions given in Fig. 19 and Fig. 20

It only makes sense to use a value, not an expression, in the label, since the corresponding hardware transition will only be concerned with values.

Hardware Domains. The upper section of Fig. 20 shows the hardware domains – that is, the states we are interested in *other* than the program. We have already described memory, store buffers, remote atomic locks and queue pairs, but note that the structures B, A, and QP are specifically maps from threads, nodes, and both, respectively, to the particular structures.

A remote atomic lock is a boolean, $\bot$ (available) or $\top$ (unavailable). A store buffer is a sequence of CPU writes and RDMA operations. A queue pair is a tuple of three sequences **pipe**, $\mathbf{wb_R}$, and $\mathbf{wb_L}$, where **pipe** may contain any of the operations described below except for a confirmation notification, $\mathbf{wb_R}$ may contain NIC remote writes and NIC remote atomic writes, and $\mathbf{wb_L}$ may contain NIC local writes and confirmation notifications.

- $y^{\overline{n}} := x^n$ denotes a put operation where the value of local memory location x is yet to be read (NIC local read);
- $y^{\overline{n}} := v$ denotes a NIC remote write of value v to remote location y, which occurs as the latter part of a put;
- $\texttt{ack}_\texttt{p}$ denotes the acknowledgement message returned by a put;
- $x^n := y^{\overline{n}}$ denotes a get operation where the value of the remote location y is yet to be read (pending NIC remote read)
- $x^n := v$ denotes a NIC local write of value v to local location x, which occurs as the latter part of a get or rRMW;
- $\texttt{RCAS}(z^n, x^{\overline{n}}, v, v')$ denotes a remote CAS towards remote location x, with expected value v, update value v', and returning to local location z;
- $\texttt{RFAA}(z^n, x^{\overline{n}}, v)$ similarly denotes a remote FAA towards x and returning to z, with increment value v;
- $y^{\overline{n}} :=_A v$ denotes a NIC remote write specifically in the case of an rRMW – it is necessary for this to be disambiguated from the NIC remote write of a put, as we will see later;
- $\texttt{rfence}(\overline{n})$ denotes a remote fence towards node $\overline{n}$;
- $\texttt{cn}$ denotes a confirmation of a successful NIC remote write.

Hardware Transitions. All remote commands enter the queue-pair pipe via the thread's store buffer. When the program takes a transition step labelled with

Program transitions: $\mathsf{Prog} \xrightarrow{\mathsf{Tid}:\mathsf{Lab}\uplus\{\varepsilon\}} \mathsf{Prog}$ **Command transitions:** $\mathsf{Comm} \xrightarrow{\mathsf{Lab}\uplus\{\varepsilon\}} \mathsf{Comm}$

$$\mathsf{Lab} \triangleq \bigcup_n \mathsf{Lab}_n \qquad l \in \mathsf{Lab}_n \triangleq \left\{ \begin{array}{l} \mathtt{lW}(x^n, v), \mathtt{lR}(x^n, v), \mathtt{CASS}(x^n, v_1, v_2), \mathtt{CASF}(x^n, v), \\ \mathtt{F}, \mathtt{P}(\overline{n}), \mathtt{Get}(x^n, y^{\overline{n}}), \mathtt{Put}(y^{\overline{n}}, x^n), \mathtt{rF}(\overline{n}), \\ \mathtt{RCAS}(y, x^{\overline{n}}, v_1, v_2), \mathtt{RFAA}(y, x^{\overline{n}}, v) \end{array} \middle| \begin{array}{l} x, y \in \mathsf{Loc}, \\ v, v_1, v_2 \in \mathsf{Val} \end{array} \right\}$$

$$\frac{\mathsf{C}_1 \xrightarrow{l} \mathsf{C}_1'}{\mathsf{C}_1; \mathsf{C}_2 \xrightarrow{l} \mathsf{C}_1'; \mathsf{C}_2} \qquad \frac{}{\mathtt{skip}; \mathsf{C} \xrightarrow{\varepsilon} \mathsf{C}} \qquad \frac{i \in \{1,2\}}{\mathsf{C}_1 + \mathsf{C}_2 \xrightarrow{\varepsilon} \mathsf{C}_i} \qquad \frac{}{\mathsf{C}^* \xrightarrow{\varepsilon} \mathtt{skip}}$$

$$\frac{}{\mathsf{C}^* \xrightarrow{\varepsilon} \mathsf{C}; \mathsf{C}^*} \qquad \frac{\mathsf{C} \rightsquigarrow \mathsf{C}'}{\mathsf{C} \xrightarrow{\varepsilon} \mathsf{C}'} \qquad \frac{\mathsf{elocs}(e) = \emptyset}{x := e \xrightarrow{\mathtt{lW}(x, [\![e]\!])} \mathtt{skip}}$$

$$\frac{\mathsf{elocs}(e_{\mathrm{old}}) = \mathsf{elocs}(e_{\mathrm{new}}) = \emptyset \qquad v \neq [\![e_{\mathrm{old}}]\!]}{z := \mathtt{CAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}) \xrightarrow{\mathtt{CASF}(x,v)} z := v}$$

$$\frac{\mathsf{elocs}(e_{\mathrm{old}}) = \mathsf{elocs}(e_{\mathrm{new}}) = \emptyset}{z := \mathtt{CAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}) \xrightarrow{\mathtt{CASS}(x, [\![e_{\mathrm{old}}]\!], [\![e_{\mathrm{new}}]\!])} z := [\![e_{\mathrm{old}}]\!]} \qquad \frac{}{\mathtt{mfence} \xrightarrow{\mathtt{F}} \mathtt{skip}}$$

$$\frac{}{x := \overline{y} \xrightarrow{\mathtt{Get}(x, \overline{y})} \mathtt{skip}} \qquad \frac{}{\overline{y} := x \xrightarrow{\mathtt{Put}(\overline{y}, x)} \mathtt{skip}}$$

$$\frac{\mathsf{elocs}(e_{\mathrm{old}}) = \mathsf{elocs}(e_{\mathrm{new}}) = \emptyset \qquad v = [\![e_{\mathrm{old}}]\!] \qquad v' = [\![e_{\mathrm{new}}]\!]}{z := \mathtt{RCAS}(\overline{x}, e_{\mathrm{old}}, e_{\mathrm{new}}) \xrightarrow{\mathtt{RCAS}(z, \overline{x}, v, v')} \mathtt{skip}}$$

$$\frac{\mathsf{elocs}(e) = \emptyset \qquad v = [\![e]\!]}{z := \mathtt{RFAA}(\overline{x}, e) \xrightarrow{\mathtt{RFAA}(z, \overline{x}, v)} \mathtt{skip}} \qquad \frac{}{\mathtt{rfence}(\overline{n}) \xrightarrow{\mathtt{rF}(\overline{n})} \mathtt{skip}} \qquad \frac{}{\mathtt{poll}(\overline{n}) \xrightarrow{\mathtt{P}(\overline{n})} \mathtt{skip}}$$

$$\frac{}{\mathtt{assume}(x = v) \xrightarrow{\mathtt{lR}(x,v)} \mathtt{skip}} \qquad \frac{v \neq v'}{\mathtt{assume}(x \neq v') \xrightarrow{\mathtt{lR}(x,v)} \mathtt{skip}} \qquad \frac{\mathsf{P}(t) \xrightarrow{l} \mathsf{C}}{\mathsf{P} \xrightarrow{t:l} \mathsf{P}[t \mapsto \mathsf{C}]}$$

$$\begin{array}{rll}
x := e & \rightsquigarrow \mathtt{assume}(y = v); x := e[v/y] & \text{for } y \in \mathsf{elocs}(e), v \in \mathsf{Val} \\
z := \mathtt{CAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}) & \rightsquigarrow \mathtt{assume}(y = v); z := \mathtt{CAS}(x, e_{\mathrm{old}}[v/y], e_{\mathrm{new}}) & \text{for } y \in \mathsf{elocs}(e_{\mathrm{old}}), v \in \mathsf{Val} \\
z := \mathtt{CAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}) & \rightsquigarrow \mathtt{assume}(y = v); z := \mathtt{CAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}[v/y]) & \text{for } y \in \mathsf{elocs}(e_{\mathrm{new}}), v \in \mathsf{Val} \\
z := \mathtt{RCAS}(\overline{x}, e_{\mathrm{old}}, e_{\mathrm{new}}) & \rightsquigarrow \mathtt{assume}(y = v); z := \mathtt{RCAS}(\overline{x}, e_{\mathrm{old}}[v/y], e_{\mathrm{new}}) & \text{for } y \in \mathsf{elocs}(e_{\mathrm{old}}), v \in \mathsf{Val} \\
z := \mathtt{RCAS}(\overline{x}, e_{\mathrm{old}}, e_{\mathrm{new}}) & \rightsquigarrow \mathtt{assume}(y = v); z := \mathtt{RCAS}(\overline{x}, e_{\mathrm{old}}, e_{\mathrm{new}}[v/y]) & \text{for } y \in \mathsf{elocs}(e_{\mathrm{new}}), v \in \mathsf{Val} \\
z := \mathtt{RFAA}(\overline{x}, e) & \rightsquigarrow \mathtt{assume}(y = v); z := \mathtt{RFAA}(\overline{x}, e[v/y]) & \text{for } y \in \mathsf{elocs}(e), v \in \mathsf{Val}
\end{array}$$

Fig. 19: The RDMA$^{\mathrm{TSO}}$ program and command transitions

a remote CAS or FAA, the hardware takes a transition with a matching label, which adds that operation to the store buffer. The seventh transition rule allows remote commands at the head of the store buffer to enter the pipe of a queue pair, determined by their target node.

So far, we have seen that when an rRMW appears in the program, we can expect there to be a hardware transition which adds it to the store buffer, and later another hardware transition which removes it from the head of the store buffer and adds it to the suitable queue pair.

The final hardware transition introduces the queue-pair transitions, indicated by $\rightarrow_{\mathtt{sqp}}$[4]. When a particular queue pair takes a transition step, involving memory and the global remote atomic lock, the hardware takes a suitable corresponding transition. The queue-pair transitions merely involve a particular subset of the hardware states, so the relationship is straightforward. This separation is purely made for clarity and simplification of the queue-pair transition rules.

Queue-Pair Transitions. From Fig. 17, recall that remote operations enter the main **pipe** of the queue pair, then are suitably processed until they exit the pipe, possibly adding a write to $\mathbf{wb_L}$ or $\mathbf{wb_R}$ (or both). Note that the pipe grows to the left, so throughout, α contains operations which are later in the program, while β contains earlier operations which have not yet completed.

The rules for remote fence, put, and get share a simple structure, where the premise for a transition either requires the operation to be at the head of the pipe $\mathbf{sqp.pipe} = \alpha \cdot (\text{operation})$, or allows it to be in the middle of the pipe $\mathbf{sqp.pipe} = \alpha \cdot (\text{operation}) \cdot \beta$, with some stipulation as to the operations allowed in β. In the prior case, the operation never executes before another, earlier operation; in the latter, it can execute before any operation in β which was issued before it. There may also be some requirement that buffers $\mathbf{wb_L}$ or $\mathbf{wb_R}$ contain no writes, due to PCIe guarantees: $\mathbf{wb_L} \in \{\mathtt{cn}\}^*$ ($\mathbf{wb_L}$ contains only confirmation notifications) or $\mathbf{wb_R} = \epsilon$ (there are no operations in $\mathbf{wb_R}$). Consider, for example, the first step of a put, which is a NIC local read described by rule 2. The value of location x is read from memory, so long as $\mathbf{wb_L}$ has no pending writes and there are no other NIC local reads earlier in the pipe.

We can then describe the rules for rfence, put, and get at a high level:

Remote fence (rule 1) An rfence may be removed from the pipe once it reaches the head (there are no earlier operations remaining to be processed). In combination with the fact that no other transition rule allows a step to be taken when there is an rfence later in the pipe, this enforces the behaviour that all remote operations prior to an rfence complete before it, and all later ones after it.

Put (rules 2-5) Rule 2: a NIC local read is performed, replacing the location x with its value in memory. Rule 3: the NIC remote write is sent to $\mathbf{wb_R}$, and an acknowledgement created in the pipe. Rule 4: the remote write is

[4] SQP stands for simplified queue pair. We only considered the simplified three-buffer queue pair, so this disambiguation is technically unnecessary, but we maintain the notation for consistency with [3]

$$\mathsf{M}\in\mathsf{Mem}\triangleq\mathsf{Loc}\to\mathsf{Val}\qquad \mathsf{B}\in\mathsf{SBMap}\triangleq\lambda t\in\mathsf{Tid}.\mathsf{SBuff}_{n(t)}$$
$$\mathsf{A}\in\mathsf{RAMap}\triangleq\lambda n.\,\{\bot,\top\}\qquad \mathsf{QP}\in\mathsf{SQPMap}\triangleq\lambda t.\big(\lambda\overline{n(t)}.\mathsf{SQPair}_n^{\overline{n}}\big)$$
$$\mathsf{b}\in\mathsf{SBuff}_n\triangleq\big\{x^n:=v,y^{\overline{n}}:=x^n,x^n:=y^{\overline{n}},\texttt{RCAS}(z^n,x^{\overline{n}},v,v'),\texttt{RFAA}(z^n,x^{\overline{n}},v),\texttt{rfence}(\overline{n})\big\}^*$$
$$\mathbf{sqp}\in\mathsf{SQPair}_n^{\overline{n}}\triangleq\mathsf{Pipe}_n^{\overline{n}}\times\mathsf{WBR}_n^{\overline{n}}\times\mathsf{WBL}_n^{\overline{n}}$$
$$\mathbf{wb_L}\in\mathsf{WBL}_n^{\overline{n}}\triangleq\big\{\mathtt{cn},x^n:=v\big\}^*\qquad \mathbf{wb_R}\in\mathsf{WBR}_n^{\overline{n}}\triangleq\big\{y^{\overline{n}}:=v,y^{\overline{n}}:=_A v\big\}^*$$
$$\mathbf{pipe}\in\mathsf{Pipe}_n^{\overline{n}}\triangleq\left\{\begin{array}{l}y^{\overline{n}}:=x^n,y^{\overline{n}}:=v,y^{\overline{n}}:=_A v,\mathtt{ack_p},x^n:=y^{\overline{n}},x^n:=v,\\ \texttt{RCAS}(z^n,x^{\overline{n}},v,v'),\texttt{RFAA}(z^n,x^{\overline{n}},v),\texttt{rfence}(\overline{n})\end{array}\right\}^*$$

$$\frac{\mathsf{B}'=\mathsf{B}[t\mapsto(x:=v)\cdot\mathsf{B}(t)]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{lW}(x,v)}\mathsf{M},\mathsf{B}',\mathsf{A},\mathsf{QP}}\qquad\frac{(\mathsf{M}\lhd\mathsf{B}(t))(x)=v}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{lR}(x,v)}\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}}$$

$$\frac{\mathsf{B}(t)=\varepsilon\qquad\mathsf{M}(x)=v_1}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{CASS}(x,v_1,v_2)}\mathsf{M}[x\mapsto v_2],\mathsf{B},\mathsf{A},\mathsf{QP}}\qquad\frac{\mathsf{B}(t)=\varepsilon\qquad\mathsf{M}(x)=v}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{CASF}(x,v)}\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}}$$

$$\frac{\mathsf{B}(t)=\varepsilon}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{F}}\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}}\qquad\frac{\mathsf{B}(t)=\mathsf{b}\cdot(x:=v)}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\varepsilon}\mathsf{M}[x\mapsto v],\mathsf{B}[t\mapsto\mathsf{b}],\mathsf{A},\mathsf{QP}}$$

$$\frac{\mathsf{B}(t)=\mathsf{b}\cdot\mathsf{rc}^n\qquad\mathsf{rc}^n\in\Big\{x:=y^{\overline{n}},y^{\overline{n}}:=x,\texttt{RCAS}(z,\overline{x},v,v'),\texttt{RFAA}(z,\overline{x},v),\texttt{rfence}(\overline{n})\Big\}\qquad\mathsf{QP}(t)(\overline{n})=\mathbf{sqp}\qquad\mathbf{sqp}'=\mathbf{sqp}[\mathbf{pipe}\mapsto\mathsf{rc}^n\cdot\mathbf{sqp}.\mathbf{pipe}]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\varepsilon}\mathsf{M},\mathsf{B}[t\mapsto\mathsf{b}],\mathsf{A},\mathsf{QP}[t\mapsto\mathsf{QP}(t)[\overline{n}\mapsto\mathbf{sqp}']]}$$

$$\frac{\mathsf{B}'=\mathsf{B}[t\mapsto(x:=\overline{y})\cdot\mathsf{B}(t)]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{Get}(x,\overline{y})}\mathsf{M},\mathsf{B}',\mathsf{A},\mathsf{QP}}\qquad\frac{\mathsf{B}'=\mathsf{B}[t\mapsto(\overline{y}:=x)\cdot\mathsf{B}(t)]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{Put}(\overline{y},x)}\mathsf{M},\mathsf{B}',\mathsf{A},\mathsf{QP}}$$

$$\frac{\mathsf{B}'=\mathsf{B}[t\mapsto\texttt{RCAS}(z,\overline{x},v,v')\cdot\mathsf{B}(t)]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{RCAS}(z,\overline{x},v,v')}\mathsf{M},\mathsf{B}',\mathsf{A},\mathsf{QP}}\qquad\frac{\mathsf{B}'=\mathsf{B}[t\mapsto\texttt{RFAA}(z,\overline{x},v)\cdot\mathsf{B}(t)]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{RFAA}(z,\overline{x},v)}\mathsf{M},\mathsf{B}',\mathsf{A},\mathsf{QP}}$$

$$\frac{\mathsf{B}'=\mathsf{B}[t\mapsto(\texttt{rfence}(\overline{n}))\cdot\mathsf{B}(t)]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{rF}(\overline{n})}\mathsf{M},\mathsf{B}',\mathsf{A},\mathsf{QP}}$$

$$\frac{\mathsf{QP}(t)(\overline{n})=\mathbf{sqp}\qquad\mathbf{sqp}.\mathbf{wb_L}=\alpha\cdot\mathtt{cn}\qquad\mathbf{sqp}'=\mathbf{sqp}[\mathbf{wb_L}\mapsto\alpha]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\mathtt{P}(\overline{n})}\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}[t\mapsto\mathsf{QP}(t)[\overline{n}\mapsto\mathbf{sqp}']]}$$

$$\frac{\mathsf{M},\mathsf{A},\mathsf{QP}(t)(\overline{n})\to_{\mathtt{sqp}}\mathsf{M}',\mathsf{A}',\mathbf{sqp}\quad\text{(Fig. 21)}}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}\xrightarrow{t:\varepsilon}\mathsf{M}',\mathsf{B},\mathsf{A}',\mathsf{QP}[t\mapsto\mathsf{QP}(t)[\overline{n}\mapsto\mathbf{sqp}]]}$$

$$\text{with}\quad(\mathsf{M}\lhd\alpha)(x)\triangleq\begin{cases}v & \text{if }\alpha=\beta\cdot(x:=v)\cdot-\wedge\forall v'.\,x:=v'\notin\beta\\ \mathsf{M}(x) & \text{if }\forall v.\,x:=v\notin\alpha\end{cases}$$

Fig. 20: RDMA$^{\text{TSO}}$ simplified hardware domains (above) and hardware transitions (below)

$$\frac{\mathbf{sqp.pipe} = \alpha \cdot (\texttt{rfence}(\overline{n}))}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\texttt{sqp}} \mathsf{M}, \mathsf{A}, \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha]}$$

$$\frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot (\overline{y} := x) \cdot \beta \quad \mathbf{wb_L} \in \{\texttt{cn}\}^* \\ \beta \in \{y' := v', y' :=_A v', \overline{y'} := v', x' := \overline{y'}, \texttt{RCAS}(z, \overline{x}, v, v'), \texttt{RFAA}(z, \overline{x}, v), \texttt{ack}_\texttt{p}\}^*\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\texttt{sqp}} \mathsf{M}, \mathsf{A}, \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha \cdot (\overline{y} := \mathsf{M}(x)) \cdot \beta]}$$

$$\frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot (\overline{y} := v) \cdot \beta \\ \beta \in \{x' := \overline{y'}, x' := v', \texttt{ack}_\texttt{p}\}^* \\ \mathbf{sqp}' = \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha \cdot \texttt{ack}_\texttt{p} \cdot \beta][\mathbf{wb_R} \mapsto (\overline{y} := v) \cdot \mathbf{sqp.wb_R}]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\texttt{sqp}} \mathsf{M}, \mathsf{A}, \mathbf{sqp}'}$$

$$\frac{\mathbf{sqp.wb_R} = \alpha \cdot (\overline{y} := v)}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\texttt{sqp}} \mathsf{M}[\overline{y} \mapsto v], \mathsf{A}, \mathbf{sqp}[\mathbf{wb_R} \mapsto \alpha]} \qquad \frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot \texttt{ack}_\texttt{p} \\ \mathbf{sqp}' = \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha][\mathbf{wb_L} \mapsto \texttt{cn} \cdot \mathbf{sqp.wb_L}]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\texttt{sqp}} \mathsf{M}, \mathsf{A}, \mathbf{sqp}'}$$

$$\frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot (x := \overline{y}) \cdot \beta \qquad \beta \in \{x' := \overline{y'}, x' := v', \texttt{ack}_\texttt{p}\}^* \\ \mathbf{sqp.wb_R} = \varepsilon \quad \mathbf{sqp}' = \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha \cdot (x := \mathsf{M}(\overline{y})) \cdot \beta]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\texttt{sqp}} \mathsf{M}, \mathsf{A}, \mathbf{sqp}'}$$

$$\frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot (x := v) \\ \mathbf{sqp}' = \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha][\mathbf{wb_L} \mapsto \texttt{cn} \cdot (x := v) \cdot \mathbf{sqp.wb_L}]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\texttt{sqp}} \mathsf{M}, \mathsf{A}, \mathbf{sqp}'}$$

$$\frac{\begin{array}{c}\mathbf{sqp.wb_L} = \alpha \cdot (x := v) \cdot \beta \quad \beta \in \{\texttt{cn}\}^* \\ \mathbf{sqp}' = \mathbf{sqp}[\mathbf{wb_L} \mapsto \alpha \cdot \beta]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\texttt{sqp}} \mathsf{M}[x \mapsto v], \mathsf{A}, \mathbf{sqp}'}$$

$$\frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot \texttt{RCAS}(z, \overline{x}, v, v') \cdot \beta \quad \mathbf{sqp.wb_R} = \epsilon \\ \mathsf{A}(n(\overline{x})) = \bot \quad \mathsf{M}(\overline{x}) \neq v \quad \beta \in \{x' := \overline{y'}, x' := v', \texttt{ack}_\texttt{p}\}^* \\ \mathbf{sqp}' = \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha \cdot (z := \mathsf{M}(\overline{x})) \cdot \beta]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\texttt{sqp}} \mathsf{M}, \mathsf{A}, \mathbf{sqp}'}$$

$$\frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot \texttt{RCAS}(z, \overline{x}, v, v') \cdot \beta \quad \mathbf{sqp.wb_R} = \epsilon \\ \mathsf{M}(\overline{x}) = v \quad \beta \in \{x' := \overline{y'}, x' := v', \texttt{ack}_\texttt{p}\}^* \\ \mathsf{A}(n(\overline{x})) = \bot \quad \mathsf{A}' = \mathsf{A}[n(\overline{x}) \mapsto \top]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\texttt{sqp}} \mathsf{M}, \mathsf{A}', \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha \cdot (z := v) \cdot (\overline{x} :=_A v') \cdot \beta]}$$

$$\frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot \texttt{RFAA}(z, \overline{x}, v) \cdot \beta \quad \mathbf{sqp.wb_R} = \epsilon \\ \mathsf{M}(\overline{x}) + v = v' \quad \beta \in \{x' := \overline{y'}, x' := v', \texttt{ack}_\texttt{p}\}^* \\ \mathsf{A}(n(\overline{x})) = \bot \quad \mathsf{A}' = \mathsf{A}[n(\overline{x}) \mapsto \top]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\texttt{sqp}} \mathsf{M}, \mathsf{A}', \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha \cdot (z := v) \cdot (\overline{x} :=_A v') \cdot \beta]}$$

$$\frac{\begin{array}{c}\mathbf{sqp.pipe} = \alpha \cdot (\overline{x} :=_A v) \cdot \beta \\ \mathbf{wb_R}' = (\overline{x} :=_A v) \cdot \mathbf{sqp.wb_R} \\ \beta \in \{x' := \overline{y'}, x' := v', \texttt{ack}_\texttt{p}\}^* \\ \mathbf{sqp}' = \mathbf{sqp}[\mathbf{pipe} \mapsto \alpha \cdot \beta][\mathbf{wb_R} \mapsto \mathbf{wb_R}']\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\texttt{sqp}} \mathsf{M}, \mathsf{A}, \mathbf{sqp}'} \qquad \frac{\begin{array}{c}\mathbf{sqp.wb_R} = \alpha \cdot (\overline{x} :=_A v) \\ \mathsf{A}' = \mathsf{A}[n(\overline{x}) \mapsto \bot] \\ \mathbf{sqp}' = \mathbf{sqp}[\mathbf{wb_R} \mapsto \alpha]\end{array}}{\mathsf{M}, \mathsf{A}, \mathbf{sqp} \rightarrow_{\texttt{sqp}} \mathsf{M}[\overline{x} \mapsto v], \mathsf{A}', \mathbf{sqp}'}$$

Fig. 21: Queue-pair transitions of the simplified RDMA$^{\text{TSO}}$ operational semantics

committed to memory once it reaches the head of the queue. Rule 5: the acknowledgement in the pipe is converted to a confirmation notification in $\mathbf{wb_L}$, so that it can be polled.

Get (rules 6-8) Rule 6: a NIC remote read replaces the location $\overline{y}$ with its value in memory. Rule 7: the NIC local write is sent to $\mathbf{wb_L}$, with a confirmation notification for the purpose of polling. Rule 8: the local write is committed to memory once there are no pending earlier writes in the queue.

Now, consider the rules for rRMWs. These rules are more complicated due to the need to check and update the remote atomic lock for the target node, which we see as $\mathsf{A}(n(\overline{x})) = \bot$ (the remote atomic lock for the target node is available), and $\mathsf{A}' = \mathsf{A}[n(\overline{x}) \mapsto \top]$ (update the remote atomic lock for the target node to indicate it is busy). We also have distinct rules for success and failure of RCAS, depending on whether the remote memory location holds the expected value ($\mathsf{M}(\overline{x}) = v$ or $\mathsf{M}(\overline{x}) \neq v$).

The rules can then be interpreted as follows:

(Rule 9) A failed RCAS read – the remote memory location does not hold the expected value. This read can only occur when the remote atomic lock is available, otherwise it would violate the atomicity guarantee. The value of $\overline{x}$ is read, and a NIC local write is added to the pipe to return that value to z. This is then handled by the same rules as for a get. The remote atomic lock is not obtained, since the remote location will not be written to.

(Rule 10) A successful RCAS read – the remote location contains the expected value. Once again, this requires that the remote atomic lock be available, and it is also obtained to ensure atomicity until the remote location is written to. A NIC local write is added to the pipe (similarly to 9), and a NIC remote atomic write to update the remote location is also added.

(Rule 11) The remote read of an RFAA – this is unconditionally successful. It is very similar to a successful RCAS, but the value for the NIC remote atomic write is calculated by adding v to the value of $\overline{x}$ in memory.

(Rule 12) A NIC remote atomic write in the pipe is processed into $\mathbf{wb_R}$ similarly to a regular NIC remote write.

(Rule 13) A NIC remote atomic write is committed to memory, and the remote atomic lock is released.

D.2 Declarative Semantics

A declarative semantics, in contrast to an operational one, describes only the events that occur in a system, not the state of the system itself. An execution is represented by a *graph*, with various relations over events. For example, given an event r, we say that it "reads-from" event w if r reads the value written to memory by w. We write $(w, r) \in \mathsf{rf}$ in this case. We then constrain these relations suitably to only allow execution graphs which make sense in the context of a program: considering rf again, we would naturally only allow $(w, r) \in \mathsf{rf}$ if the values of the read and write match.

Then, we know that an execution of a program is allowed if the graph of the execution is consistent. Contrast this with the operational semantics: there, our guarantee comes from the individual transition rules; here, it is due to the overall structure of the graph.

Events and Executions. An *execution* is a graph comprising a set of events and several relations over events; events are represented as graph nodes, and the relations are edges. An event has a unique identifier ι, is created by a thread $t \in \mathsf{Tid}$, and has an event label $l \in \mathsf{ELab}$ which describes the event.

Definition 17 (Labels and events). *Each event label is associated to a node n. The set of* event labels of node n *is denoted by $l \in \mathsf{ELab}_n$, where l is a tuple with one of the following forms:*

- *(CPU) local read:* $l = \mathtt{lR}(x^n, v_r)$
- *(CPU) local write:* $l = \mathtt{lW}(x^n, v_w)$
- *(CPU) CAS:* $l = \mathtt{CAS}(x^n, v_r, v_w)$
- *(CPU) memory fence:* $l = \mathtt{F}$
- *(CPU) poll:* $l = \mathtt{P}(\overline{n})$
- *NIC local read:* $l = \mathtt{nlR}(x^n, v_r, \overline{n})$
- *NIC remote write:* $l = \mathtt{nrW}(y^{\overline{n}}, v_w)$
- *NIC remote read:* $l = \mathtt{nrR}(y^{\overline{n}}, v_r)$
- *NIC local write:* $l = \mathtt{nlW}(x^n, v_w, \overline{n})$
- *NIC fence:* $l = \mathtt{nF}(\overline{n})$
- *NIC atomic remote read:* $l = \mathtt{narR}(y^{\overline{n}}, v_r)$
- *NIC atomic remote write:* $l = \mathtt{narW}(y^{\overline{n}}, v_w)$

The set of event labels are defined $\mathsf{ELab} \triangleq \bigcup_n \mathsf{ELab}_n$.
An event, $\mathsf{e} \in \mathsf{Event}$, *is a triple (ι, t, l), where $\iota \in \mathbb{N}$, $t \in \mathsf{Tid}$ and $l \in \mathsf{ELab}_{n(t)}$.*

We distinguish between events associated with the CPU (left) or NIC (right), with the prefix `n` used for all NIC event labels. Note that a put, get, or rRMW is modelled by multiple events: a put $\overline{x} := y$ comprises a NIC local read event of type `nlR` (on y) followed by a NIC remote write event `nrW` (on $\overline{x}$); conversely a get $x := \overline{y}$ comprises events of type `nrR` (on $\overline{y}$) and `nlW` (on x). A successful rRMW (either successful RCAS or RFAA) is modelled by three events of type `narR`, `narW` and `nlW`, while a failed rRMW (RCAS only) is modelled by only `narR` and `nlW`.

For a given label l, we write $\mathsf{type}(l)$, $\mathtt{loc}(l)$, $v_\mathrm{r}(l)$, $v_\mathrm{w}(l)$, $n(l)$ and $\overline{n}(l)$ for the type, location, value read or written, and local or remote node, where applicable. For example, consider $l = \mathtt{nlR}(x^n, v_\mathrm{r}, \overline{n})$:

- $\mathsf{type}(\mathtt{nlR}(x^n, v_\mathrm{r}, \overline{n})) = \mathtt{nlR}$
- $\mathtt{loc}(\mathtt{nlR}(x^n, v_\mathrm{r}, \overline{n})) = x$
- $v_\mathrm{r}(\mathtt{nlR}(x^n, v_\mathrm{r}, \overline{n})) = v_\mathrm{r}$
- $v_\mathrm{w}(\mathtt{nlR}(x^n, v_\mathrm{r}, \overline{n}))$ is undefined
- $n(\mathtt{nlR}(x^n, v_\mathrm{r}, \overline{n})) = n$
- $\overline{n}(\mathtt{nlR}(x^n, v_\mathrm{r}, \overline{n})) = \overline{n}$

We write $\iota(\mathsf{e})$, $t(\mathsf{e})$, $l(\mathsf{e})$ for the relevant constituents of an event tuple $\mathsf{e} = (\iota, t, l)$. We lift the functions on event labels to functions on events, for example $\mathsf{type}(\mathsf{e}) \triangleq \mathsf{type}(l(\mathsf{e}))$.

Difference with MOWGLI**.** The labels of this declarative semantics (à la [3]) roughly corresponds to the stamps of MOWGLI (à la [4]) in the main paper. E.g.

a NIC local read has a label `nlR` here and corresponds to the stamp (family) `aNLR` in Fig. 9. However, there is one major discrepancy. The declarative semantics of this section distinguishes between NIC remote writes performed by put operations (label `nrW`) and performed by rRMW operations (label `narW`), while they both correspond to the single stamp $\texttt{aNRW}_n$.

The main reason is that this semantics, by decomposing operations into multiple events, creates a po ordering between the remote write and local write parts of a (successful) remote RMW. As such, we cannot enforce a ppo ordering between the two parts (`narW` and `nlW`) as they might not finish in order, but we can enforce a ppo ordering between the remote write of a put and later local writes (`nrW` and `nlW`), making the semantics more straightforward. With MOWGLI, each operation generates a *single* event, and there is no po ordering between subevents of the same event. Thus we can add a dependency between $\texttt{aNRW}_n$ and $\texttt{aNLW}_n$ (cell G10 in Fig. 9), and it will not create an internal dependency within the same rRMW operation.

A secondary reason is that the two labels (`nrW` and `narW`) correspond to different behaviours of the operational semantics. Making the distinction renders the equivalence proof more tractable.

Issue and Observation Points. Some types of events do not occur instantaneously: for example, a local write event `lW` first enters the store before, before later being committed to memory. We therefore distinguish between the point at which an event is *issued* by the CPU or NIC, and the point at which it is *observed*, when its effect becomes visible in memory. An event is *instantaneous* if it either has no visible effect on memory, or if it affects memory immediately, as is the case for a local CAS operation. For instantaneous events, the issue and observation points coincide.

Notation. Once again, we follow and extend the notation of [3]. For a set A and relations r, r_1, r_2, we write:

- $\boldsymbol{r^+}$ for the transitive closure of r;
- $\boldsymbol{r^{-1}}$ for the inverse of r;
- $\boldsymbol{r|_A} \triangleq r \cap (A \times A)$ for the restriction of r to set A;
- $\boldsymbol{[A]} \triangleq \{(a,a) \mid a \in A\}$ for the identity relation
- $\boldsymbol{r_1; r_2} \triangleq \{(a,b) \mid \exists c.(a,c) \in r_1 \wedge (c,b) \in r_2\}$ for relational composition;
- $\boldsymbol{r|_{\mathbf{imm}}} \triangleq r \setminus (r;r)$ for the immediate edges in r, when r is a strict partial order.

For a set of events E, location x and label type `X`, we also define:

- $E_x \triangleq \{\texttt{e} \in E \mid \texttt{loc(e)} = x\}$, the events towards x;
- $E.\texttt{X} \triangleq \{\texttt{e} \in E \mid \texttt{type(e)} = \texttt{X}\}$, the events of type `X`;
- $E.\mathcal{R} \triangleq E.\texttt{lR} \cup E.\texttt{CAS} \cup E.\texttt{nlR} \cup E.\texttt{nrR} \cup E.\texttt{narR}$, the set of reads;
- $E.\mathcal{W} \triangleq E.\texttt{lW} \cup E.\texttt{CAS} \cup E.\texttt{nlW} \cup E.\texttt{nrW} \cup E.\texttt{narW}$, the set of writes;
- $E.\texttt{Inst} \triangleq E \setminus (E.\texttt{lW} \cup E.\texttt{nlW} \cup E.\texttt{nrW} \cup E.\texttt{narW})$, the set of instantaneous events.

Finally, we define the following relations:

Same-location: $\mathsf{sloc} \triangleq \{(\mathsf{e},\mathsf{e}') \in \mathsf{Event}^2 \mid \mathtt{loc}(\mathsf{e}) = \mathtt{loc}(\mathsf{e}')\}$
Same-thread: $\mathsf{sthd} \triangleq \{(\mathsf{e},\mathsf{e}') \in \mathsf{Event}^2 \mid t(\mathsf{e}) = t(\mathsf{e}')\}$
Same-queue-pair: $\mathtt{sqp} \triangleq \{(\mathsf{e},\mathsf{e}') \in \mathsf{Event}^2 \mid t(\mathsf{e}) = t(\mathsf{e}') \wedge \overline{n}(\mathsf{e}) = \overline{n}(\mathsf{e}')\}$

Note that these relations are all symmetric, and $\mathtt{sqp} \subseteq \mathsf{sthd}$. Given events E, we write $E.\mathsf{sloc}$ for $\mathsf{sloc}|_E$, likewise for $E.\mathsf{sthd}$ and $E.\mathtt{sqp}$.

Definition 18 (Pre-executions). *A* pre-execution *is a tuple* $G = \langle E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{pf}, \mathsf{nfo}, \mathsf{rao}\rangle$*, where:*

- $E \subseteq \mathsf{Event}$ *is the set of events and includes a set of* initialisation *events,* $E^0 \subseteq E$*, comprising a single write with label* $\mathtt{lW}(x, 0)$ *for each* $x \in \mathsf{Loc}$*.*
- $\mathsf{po} \subseteq E \times E$ *is the '*program order*' relation defined as a disjoint union of strict total orders, each ordering the events of one thread, with* $E^0 \times (E \setminus E^0) \subseteq \mathsf{po}$*.*
- $\mathsf{rf} \subseteq E.\mathcal{W} \times E.\mathcal{R}$ *is the '*reads-from*' relation on events of the same location with matching values; i.e.* $(a, b) \in \mathsf{rf} \Rightarrow (a, b) \in \mathsf{sloc} \wedge v_w(a) = v_r(b)$*. Moreover,* rf *is total and functional on its range: every read in* $E.\mathcal{R}$ *is related to exactly one write in* $E.\mathcal{W}$*.*
- $\mathsf{mo} \triangleq \bigcup_{x \in \mathsf{Loc}} \mathsf{mo}_x$ *is the '*modification-order*', where each* mo_x *is a strict total order on* $E.\mathcal{W}_x$ *with* $E^0_x \times (E.\mathcal{W}_x \setminus E^0_x) \subseteq \mathsf{mo}_x$ *describing the order in which writes on* x *reach the memory.*
- $\mathsf{pf} \subseteq (E.\mathtt{nlW} \cup E.\mathtt{nrW}) \times E.\mathtt{P}$ *is the '*polls-from*' relation, relating earlier (in program-order) NIC writes to later poll operations on the* same queue pair*; i.e.* $\mathsf{pf} \subseteq \mathsf{po} \cap \mathtt{sqp}$*. Moreover,* pf *is functional on its domain (every NIC write can be be polled at most once), and* pf *is total and functional on its range (every poll in* $E.\mathtt{P}$ *polls from exactly one NIC write).*
- $\mathsf{nfo} \subseteq E.\mathtt{sqp}$ *is the '*NIC flush order*', such that for all* $(a, b) \in E.\mathtt{sqp}$*, if* $a \in E.\mathtt{nlR}, b \in E.\mathtt{nlW}$*, then* $(a, b) \in \mathsf{nfo} \cup \mathsf{nfo}^{-1}$*, and if* $a \in (E.\mathtt{nrR} \cup E.\mathtt{narR}), b \in (E.\mathtt{nrW} \cup E.\mathtt{narW})$*, then* $(a, b) \in \mathsf{nfo} \cup \mathsf{nfo}^{-1}$*.*
- $\mathsf{rao} \triangleq \bigcup_{n \in \mathsf{Node}} \mathsf{rao}_n$ *is the '*remote-atomic-order*', where each* rao_n *is a strict total order on* $\{e \mid e \in E.\mathtt{narR} \wedge \overline{n}(e) = n\}$ *describing the order in which remote atomics towards* n *are executed.*

The definitions of po, rf and mo are familiar from TSO, while pf and nfo are introduced in [3]. As mentioned previously, nfo represents the PCIe guarantee that a NIC local read flushes pending NIC remote writes on the same queue pair, and likewise for NIC local reads/writes. We introduce rao, which totally orders NIC remote atomic reads towards a given node and help enforce the rRMW atomicity guarantee.

Derived Relations. Given a pre-execution $\langle E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{pf}, \mathsf{nfo}, \mathsf{rao}\rangle$, we define the following *derived* relations:

- $\mathsf{rb} \triangleq (\mathsf{rf}^{-1}; \mathsf{mo}) \setminus [E]$ is the *reads-before* relation, relating each read r to writes that are mo-after the write from which r reads.

- $\mathsf{rf_i} \triangleq [\mathtt{lW}]; (\mathsf{rf} \cap \mathsf{sthd}); [\mathtt{lR}]$ is the rf-*buffer* relation, which includes rf edges only for CPU operations on the same thread, which thus share a store buffer; therefore when $w \xrightarrow{\mathsf{rf_i}} r$, it may be that the write w is not yet visible (committed to memory) when it is read by r, since CPU reads check the store buffer.
- $\mathsf{rf_e} \triangleq \mathsf{rf} \setminus \mathsf{rf_i}$ is the $\mathsf{rf_i}$-complement: if $w \xrightarrow{\mathsf{rf_e}} r$, then r only occurs after w is observable.
- $\mathsf{rb_i} \triangleq [\mathtt{lR}]; (\mathsf{rb} \cap \mathsf{sthd}); [\mathtt{lW}]$ is the rb-*buffer* relation, analogously.
- $\mathsf{ar} \triangleq [\mathtt{narW}]; (\mathsf{po}|_{\mathrm{imm}}{}^{-1})$ is the *atomic-write-to-read* relation, connecting the remote write of a successful rRMW to their corresponding read.

Note that these derived relations contain no additional information. We introduce them for ease and brevity of notation.

Preserved Program Order. We identify which events in po are *issued* in order, and which are *observed* in order. The observation point of an event is no earlier than its issue point, so two events in po are only observed in order if they are issued in order. Furthermore, when the po-earlier event is instantaneous, the events are observed in order if and only if they are issued in order.

We therefore define two relations: ippo, the *issue-preserved-program-order* relation, and oppo, the *observation-preserved-program-order* relation, where $\mathsf{oppo} \subseteq \mathsf{ippo} \subseteq \mathsf{po}$. The tables in Fig. 22 show these relations. Each row indicates the po-earlier event, while each column indicates that which is po-later. A cell labelled ✓ indicates the event pair is in ippo (resp. oppo) and must be issued (resp. observed) in program order, while ✗ indicates they are not in ippo/oppo and may be issued/observed out of program order. The label `sqp` indicates that the events are in ippo/oppo if they are events on the same queue pair.

We can observe high-level reordering rules by looking at each quadrant of the two tables, which partition the event pairs by their categorisation as CPU or NIC events. The top left quadrant contains pairs of CPU events. Observe that CPU events are always issued in program order, and only an earlier CPU write may be observed out of order, as all other CPU events are instantaneous. The bottom left quadrants shows that an earlier NIC event may always be issued or observed after a later CPU event, matching our intuition that NIC events execute concurrently, as if in a separate thread; conversely the top right shows that earlier CPU events always complete before later NIC events. In the bottom right quadrant, we can see that a pair of NIC events are only ordered if they are on the same queue pair.

The relations ippo and oppo differ in only six cells. A CPU write may be buffered and hence not observed by a later CPU read or poll (B1 and B5). Other CPU writes and CAS or fence operations go via the store buffer, so earlier writes will be observed first. Similarly, a remote fence may be observed before an earlier NIC remote (atomic) write (resp. local), if that write is buffered in $\mathbf{wb_R}$ (resp. $\mathbf{wb_L}$) (G12 and I12, resp. K12). Finally, a po-later `nlW` may be observed before a po-earlier `narW` (I11). This occurs specifically in the case where both are created by the same rRMW, because the writes are sent to $\mathbf{wb_L}$ and $\mathbf{wb_R}$ respectively and may be committed in either order.

Later in Program Order

ippo			CPU					NIC						
			1	2	3	4	5	6	7	8	9	10	11	12
			lR	lW	CAS	F	P	nlR	nrW	narR	narW	nrR	nlW	nF
Earlier in Program Order	CPU	A lR	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		B lW	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		C CAS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		D F	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		E P	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	NIC	F nlR	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	sqp	sqp
		G nrW	✗	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	sqp
		H narR	✗	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	sqp
		I narW	✗	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	sqp
		J nrR	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	sqp	sqp
		K nlW	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	sqp	sqp
		L nF	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	sqp	sqp

Later in Program Order

oppo			CPU					NIC						
			1	2	3	4	5	6	7	8	9	10	11	12
			lR	lW	CAS	F	P	nlR	nrW	narR	narW	nrR	nlW	nF
Earlier in Program Order	CPU	A lR	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		B lW	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
		C CAS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		D F	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		E P	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	NIC	F nlR	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	sqp	sqp
		G nrW	✗	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	✗
		H narR	✗	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	sqp
		I narW	✗	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	✗	✗
		J nrR	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	sqp	sqp
		K nlW	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	sqp	✗
		L nF	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp	sqp	sqp

Fig. 22: The RDMA$^{\text{TSO}}$ ordering constraints on ippo (above) and oppo (below), where ✓ denotes that instructions are ordered (and cannot be reordered), ✗ denotes they are not ordered (and may be reordered), and sqp denotes they are ordered iff they are on the same queue pair.

Definition 19 (Executions). *A pre-execution* $G = \langle E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{pf}, \mathsf{nfo}, \mathsf{rao} \rangle$ *is* well-formed *if the following hold for all* w, r, w_1, w_2, p_2*:*

1) *Poll events poll-from the oldest non-polled remote operation on the same queue pair:*
   if $w_1 \in G.\mathtt{nlW} \cup G.\mathtt{nrW}$ *and* $w_1 \xrightarrow{\mathsf{po} \cap \mathsf{sqp}} w_2 \xrightarrow{\mathsf{pf}} p_2$*, then there exists* p_1 *such that* $w_1 \xrightarrow{\mathsf{pf}} p_1 \xrightarrow{\mathsf{po}} p_2$*.*
2) *Each put (resp. get) operation corresponds to two events: a read and a write with the read immediately preceding the write in* po*: 1) if* $r \in G.\mathtt{nlR}$ *(resp.* $r \in G.\mathtt{nrR}$*), then* $(r, w) \in \mathsf{po}|_{imm}$ *for some* $w \in G.\mathtt{nrW}$ *(*$w \in G.\mathtt{nlW}$*); and 2) if* $w \in G.\mathtt{nrW}$ *then* $(r, w) \in \mathsf{po}|_{imm}$ *for some* $r \in G.\mathtt{nlR}$*. The case* $w \in G.\mathtt{nlW}$ *is handled by (6) below.*
3) *Read and write events of a put (resp. get) have matching values:*
   if $(r, w) \in G.\mathsf{po}|_{imm}$*,* $\mathsf{type}(r) \in \{\mathtt{nlR}, \mathtt{nrR}\}$ *and* $\mathsf{type}(w) \in \{\mathtt{nlW}, \mathtt{nrW}\}$*, then* $v_r(r) = v_w(w)$*.*
4) *Each rRMW operation corresponds to either an atomic remote read followed by a local write, or an atomic remote read, followed by an atomic remote write, followed by a local write: 1) if* $r \in G.\mathtt{narR}$ *then* $(r, w_1) \in \mathsf{po}|_{imm}$ *for some* $w_1 \in G.\mathtt{narW} \cup G.\mathtt{nlW}$*, and if* $w_1 \in G.\mathtt{narW}$ *then* $(w_1, w_2) \in \mathsf{po}|_{imm}$ *for some* $w_2 \in G.\mathtt{nlW}$*, and 2) if* $w_1 \in G.\mathtt{narW}$ *then* $(r, w_1) \in \mathsf{po}|_{imm}$ *for some* $r \in \mathtt{narR}$*, and* $(w_1, w_2) \in \mathsf{po}|_{imm}$ *for some* $w_2 \in \mathtt{nlW}$*. The case for* $w_2 \in \mathtt{nlW}$ *is handled by (6) below.*
5) *Remote atomic read and local write events of an rRMW have matching values: if* $(r, w) \in G.\mathsf{po}|_{imm}$*,* $\mathsf{type}(r) = \mathtt{narR}$ *and* $\mathsf{type}(w) = \mathtt{nlW}$*, then* $v_r(r) = v_w(w)$*; and if* $(r, w_1), (w_1, w_2) \in G.po|_{imm}$*,* $\mathsf{type}(r) = \mathtt{narR}$*,* $\mathsf{type}(w_1) = \mathtt{narW}$ *and* $\mathsf{type}(w_2) = \mathtt{nlW}$*, then* $v_r(r) = v_w(w_2)$*.*
6. *(2) and (4) auxiliary in the case of* $w \in \mathtt{nlW}$*. If* $w \in G.\mathtt{nlW}$ *then either:*
   1) $(r, w) \in \mathsf{po}|_{imm}$ *for some* $r \in G.\mathtt{nrR}$ *or*
   2) $(r, w) \in \mathsf{po}|_{imm}$ *for some* $r \in G.\mathtt{narR}$ *or*
   3) $(r, w'), (w', w) \in \mathsf{po}|_{imm}$ *for some* $r \in G.\mathtt{narR}$ *and* $w' \in G.\mathtt{narW}$*.*

An execution *is a pre-execution (Def. 18) that is well-formed.*

Given an execution G, we write $G.E$, $G.\mathsf{mo}$, $G.\mathsf{ippo}$ and so forth to project the components and derived relations of G. When the execution is question is clear, we simply write E, mo or similar.

Definition 20 (RDMA$^{\text{TSO}}$-consistency). *An execution* $\langle E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{pf}, \mathsf{nfo}, \mathsf{rao} \rangle$ *is* RDMA$^{\text{TSO}}$-consistent *iff 1)* ib *is irreflexive; and 2)* ob *is irreflexive, where:*

$$\mathsf{ib} \triangleq \left(\mathsf{ippo} \cup \mathsf{rf} \cup \mathsf{pf} \cup \mathsf{nfo} \cup \mathsf{rb}_{\mathsf{i}} \cup (\mathsf{ob}; [\mathtt{Inst}])\right)^{+} \qquad \text{('issued-before')}$$

$$\mathsf{ob} \triangleq \left(\mathsf{oppo} \cup \mathsf{rf}_{\mathsf{e}} \cup ([\mathtt{nlW}]; \mathsf{pf}) \cup \mathsf{nfo} \cup \mathsf{rb} \cup \mathsf{mo} \cup \mathsf{rao} \cup (\mathsf{ar}; \mathsf{rao}) \cup ([\mathtt{Inst}]; \mathsf{ib})\right)^{+} \qquad \text{('observed-before')}$$

These relations extend ippo and oppo respectively to describe the issue and observation orders across threads and nodes. They are required to be irreflexive, i.e. an event cannot be issued or observed before itself.

The remaining components of ib are (a) rf: if $w \xrightarrow{\mathsf{rf}} r$ then w was at least issued (if not observed) before r – recall that if the read and write are both CPU events on the same thread, w may not be observable; (b) pf: similarly $w \xrightarrow{\mathsf{pf}} p$ only if w was issued before p; (c) nfo: NIC events arrive in $\mathbf{wb_L}/\mathbf{wb_R}$ in the order they are issued; (d) $\mathsf{rb_i}$: if $r \xrightarrow{\mathsf{rb_i}} w$, then r must be issued before w, otherwise r would read from w or an mo-later w'; (e) ob; [Inst]: in general, an event is observed no earlier than it is issued, and for an instantaneous event, the two points coincide. Thus $e \xrightarrow{\mathsf{ob}} e'$ implies $e \xrightarrow{\mathsf{ib}} e'$ when e' is instantaneous. As noted in [3], this last component is optional and does not modify the semantics.

On the other hand, for ob we have (a) $\mathsf{rf_e}$: if $w \xrightarrow{\mathsf{rf_e}} r$ then w was committed to memory before r, since r cannot read from the store buffer of another thread; (b) [nlW]; pf: NIC local writes cannot be polled until they are committed to memory; (c) nfo: NIC events are observed in the same order they arrive in $\mathbf{wb_L}/\mathbf{wb_R}$; (d) rb: if $r \xrightarrow{\mathsf{rb}} w$, then w was not observed before r, otherwise it would have been committed to memory before r; (e) mo: if $w \xrightarrow{\mathsf{mo}} w'$, then w was observed in memory before w'; (f) rao: remote atomic reads are (issued and) completed in the defined order; (g) ar; rao: if $w \xrightarrow{\mathsf{ar}} r \xrightarrow{\mathsf{rao}} r'$, then we have that r and w are the read and write of the same rRMW operation, thus w must be observed before the rao-later r' to ensure atomicity. (h) [Inst]; ib: by a similar logic to above, we know that the ib-earlier instantaneous event is also observed earlier, since its issue and observation points coincide.

Semantics of a Program. Given a program P, we can generate an event graph (E, po), by a standard process, which we describe below. We then choose any rf, mo, pf, nfo, rao such that the execution is consistent. The semantics of P are the set of consistent executions of P.

Thread to Event Graph. Given a thread identifier $t \in \mathsf{Tid}$ and a sequence of labels $l_1, \ldots, l_n \in \mathsf{ELab}$, we define the *event graphs of* t as $(\{e_1, \ldots, e_n\}, \mathsf{po}) \in G^t(l_1, \ldots, l_n)$ where: (a) $l(e_i) = l_i$ for all $1 \leq i \leq n$; (b) $\iota(e_i) \neq \iota(e_j)$ for all $1 \leq i < j \leq n$; (c) $t(e_i) = t$ for all $1 \leq i \leq n$; (d) $\mathsf{po} = \{(e_i, e_j) \mid 1 \leq i < j \leq n\}$.

Initial Event Graph. Given a set of locations Loc, we define $G_{init} = (E_0, \emptyset)$, such that for each $x \in \mathsf{Loc}$ there is exactly one $e \in E_0$ with $l(e) = \mathtt{lW}(x, 0)$, and every event in E_0 has a unique identifier. We call E_0 the set of *initialisation events*.

Sequential Composition. For two event graphs G_1 and G_2, we define their *sequential* composition $G_1; G_2 = (E, \mathsf{po})$ where

$$E \triangleq G_1.E \uplus G_2.E$$
$$\mathsf{po} \triangleq G_1.\mathsf{po} \cup G_2.\mathsf{po} \cup (G_1.E \times G_2.E)$$

Note that all events in G_2 are ordered po-after every event in G_1. Sequential composition is defined only where the set of events of each graph are disjoint, i.e. $G_1.E \cap G_2.E = \emptyset$.

Parallel Composition. We define *parallel* composition by $G_1 \parallel G_2 = (E, \mathsf{po})$ where

$$E \triangleq G_1.E \uplus G_2.E$$
$$\mathsf{po} \triangleq G_1.\mathsf{po} \cup G_2.\mathsf{po}$$

Note that the events of each graph are not po-ordered with respect to one another. We also require that the event sets be disjoint. As this operation is commutative and associative, it is straightforward to lift it to sets of graphs, which we denote by $\parallel \mathcal{G}$, where $\mathcal{G}$ is a set of event graphs.

Program to Event Graph. A program P *generates* G if $G = G_{init}; (\|_{t \in \mathsf{Tid}}\ G_t)$ and there is a set of sequences $s_t \in S$ such that $\mathsf{P}(t) \rightarrowtail s_t$ and $G_t \in G^t(s_t)$ for all $t \in \mathsf{Tid}$.

The operation $C \rightarrowtail s$ relates a sequential program C to a sequence of labels s it generates. The definition is standard and show in Fig. 23. Note that RDMA operations generate multiple events, and for local and remote CAS operations, we distinguish between success and failure cases.

Theorem 7. *The operational and declarative semantics of* $\mathrm{RDMA}^{\mathrm{TSO}}_{\mathrm{RMW}}$ *are equivalent.*

Proof. See Appendix D.3 onwards, extending the proof of [3].

D.3 Annotated Labels and Inference Rules

On top of the 12 labels presented in Appendix D.2, we create six new labels: $\mathtt{Put}(\overline{y}, x)$, $\mathtt{Get}(x, \overline{y})$, $\mathtt{RCAS}(z, \overline{x}, v, u)$, $\mathtt{RFAA}(z, \overline{x}, u)$, $\mathtt{nlEX}(\overline{n})$, and $\mathtt{nrEX}(\overline{n})$. These labels can also be used to create events (when bundled with an event identifier and a thread identifier).

We note E^{ext} the extended set of all events, including the six new labels.

Recall that $\mathcal{R} = \mathtt{lR} \cup \mathtt{CAS} \cup \mathtt{nlR} \cup \mathtt{nrR} \cup \mathtt{narR} \subseteq E^{\mathrm{ext}}$ and $\mathcal{W} = \mathtt{lW} \cup \mathtt{CAS} \cup \mathtt{nlW} \cup \mathtt{nrW} \cup \mathtt{narW} \subseteq E^{\mathrm{ext}}$. We also note $\mathtt{nEX} = \mathtt{nlEX} \cup \mathtt{nrEX}$ and $\mathtt{rRMW} = \mathtt{RCAS} \cup \mathtt{RFAA}$.

For annotated labels, we reuse most names from labels, but they are different entities. For instance we note $r \in \mathtt{lR}$ for an event with label $\mathtt{lR}$, while $\lambda = \mathtt{lR}\langle \ldots \rangle$ is an annotated label.

We use $\mathsf{type}(\lambda)$ to denote the type of the annotated label ($\mathtt{lR}$, $\mathtt{lW}$, $\mathtt{CAS}$, $\mathtt{F}$, $\mathtt{Push}$, $\mathtt{NIC}$, $\mathtt{nlR}$, $\mathtt{nrR}$, $\mathtt{nlW}$, $\mathtt{nrW}$, CN, $\mathtt{P}$, $\mathtt{nF}$, B, $\mathcal{E}$). We use $r(\lambda), w(\lambda), u(\lambda), a(\lambda), f(\lambda), p(\lambda), e(\lambda), \ldots$ to access the elements of a $\lambda \in \mathsf{ALabel}$ where applicable. Also, we note $t(\lambda)$ for the thread of the first argument of λ.

The annotated program transitions (Fig. 25) use an additional annotated label $\mathtt{CASF}\langle r, w \rangle$ with $r \in \mathtt{lR}$ and $w \in \mathcal{W}$ to represent a failed CAS operation. This case is then translated into two labels (a memory fence and a local read) when creating a path in §D.4. Also, note that the annotated domains (e.g. the store buffers and the queue pairs) contain events, not annotated labels.

$$\frac{C \rightsquigarrow C' \qquad C' \rightarrowtail s}{C \rightarrowtail s} \qquad \frac{C_1 \rightarrowtail s_1 \qquad C_2 \rightarrowtail s_2}{C_1; C_2 \rightarrowtail s_1, s_2} \qquad \frac{\mathsf{elocs}(e) = \emptyset}{x := e \rightarrowtail \mathtt{lW}(x, [\![e]\!])}$$

$$\frac{\mathsf{elocs}(e_{\mathrm{old}}) = \mathsf{elocs}(e_{\mathrm{new}}) = \emptyset \qquad z := [\![e_{\mathrm{old}}]\!] \rightarrowtail s}{z := \mathtt{CAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}) \rightarrowtail \mathtt{CAS}(x, [\![e_{\mathrm{old}}]\!], [\![e_{\mathrm{new}}]\!]), s}$$

$$\frac{\mathsf{elocs}(e_{\mathrm{old}}) = \mathsf{elocs}(e_{\mathrm{new}}) = \emptyset \qquad v \neq [\![e_{\mathrm{old}}]\!] \qquad z := v \rightarrowtail s}{z := \mathtt{CAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}) \rightarrowtail \mathtt{F}, \mathtt{lR}(x, v), s} \qquad \frac{}{\mathtt{mfence} \rightarrowtail \mathtt{F}}$$

$$\frac{}{\mathtt{assume}(x = v) \rightarrowtail \mathtt{lR}(x, v)} \qquad \frac{v' \neq v}{\mathtt{assume}(x \neq v) \rightarrowtail \mathtt{lR}(x, v')}$$

$$\frac{}{x := y^{\overline{n}} \rightarrowtail \mathtt{nrR}(y^{\overline{n}}, v), \mathtt{nlW}(x, v, \overline{n})} \qquad \frac{}{y^{\overline{n}} := x \rightarrowtail \mathtt{nlR}(x, v, \overline{n}), \mathtt{nrW}(y^{\overline{n}}, v)}$$

$$\frac{}{\mathtt{rfence}(\overline{n}) \rightarrowtail \mathtt{nF}(\overline{n})} \qquad \frac{}{\mathtt{poll}(\overline{n}) \rightarrowtail \mathtt{P}(\overline{n})} \qquad \frac{}{\mathtt{skip} \rightarrowtail \epsilon}$$

$$\frac{\mathsf{elocs}(e_{old}) = \mathsf{elocs}(e_{new}) = \emptyset \qquad v \neq [\![e_{old}]\!]}{z := \mathtt{RCAS}(x^{\overline{n}}, e_{old}, e_{new}) \rightarrowtail \mathtt{narR}(x^{\overline{n}}, v), \mathtt{nlW}(z, v, \overline{n})}$$

$$\frac{\mathsf{elocs}(e_{old}) = \mathsf{elocs}(e_{new}) = \emptyset}{z := \mathtt{RCAS}(x^{\overline{n}}, e_{old}, e_{new}) \rightarrowtail \mathtt{narR}(x^{\overline{n}}, [\![e_{old}]\!]), \mathtt{narW}(x^{\overline{n}}, [\![e_{new}]\!]), \mathtt{nlW}(z, [\![e_{old}]\!], \overline{n})}$$

$$\frac{\mathsf{elocs}(e) = \emptyset \qquad v' = v + [\![e]\!]}{z := \mathtt{RFAA}(x^{\overline{n}}, e) \rightarrowtail \mathtt{narR}(x^{\overline{n}}, v), \mathtt{narW}(x^{\overline{n}}, v'), \mathtt{nlW}(z, v, \overline{n})}$$

Fig. 23: Label Sequences Construction

initialisation. Given a program P, let

$\mathsf{M}_0 \in \mathsf{AMem}$ s.t. $\forall x \in \mathsf{Loc}.\ \mathsf{M}_0(x) = init_x$ with $l(init_x) \triangleq \mathtt{1W}(x, 0)$
$\mathsf{b}_0 \in \mathsf{ASBuff}$ $\mathsf{b}_0 \triangleq \varepsilon$
$\mathsf{B}_0 \in \mathsf{ASBMap}$ $\mathsf{B}_0 \triangleq \lambda t.\mathsf{b}_0$
$\mathsf{A}_0 \in \mathsf{RAMap}$ $\mathsf{A}_0 \triangleq \lambda t.\bot$
$\mathsf{qp}_0 \in \mathsf{AQPair}$ $\mathsf{qp}_0 \triangleq \langle \varepsilon, \varepsilon, \varepsilon \rangle$
$\mathsf{QP}_0 \in \mathsf{AQPMap}$ $\mathsf{QP}_0 \triangleq \lambda t.\lambda n.\mathsf{qp}_0$

$\lambda \in \mathsf{ALabel}$

$$
\begin{array}{lll}
\lambda \triangleq & \mid \mathtt{lR}\langle r, w\rangle & \text{where } r \in \mathtt{lR}, w \in \mathcal{W}, \mathsf{eq}_{\mathtt{loc\&v}}(r, w) \\
& \mid \mathtt{lW}\langle w\rangle & \text{where } w \in \mathtt{lW} \\
& \mid \mathtt{CAS}\langle u, w\rangle & \text{where } u \in \mathtt{CAS}, w \in \mathcal{W}, \mathsf{eq}_{\mathtt{loc\&v}}(u, w) \\
& \mid \mathtt{F}\langle f\rangle & \text{where } f \in \mathtt{F} \\
& \mid \mathtt{Push}\langle a\rangle & \text{where } a \in (\mathtt{Put} \cup \mathtt{Get} \cup \mathtt{RCAS} \cup \mathtt{RFAA} \cup \mathtt{nF}) \\
& \mid \mathtt{NIC}\langle a\rangle & \text{where } a \in (\mathtt{Put} \cup \mathtt{Get} \cup \mathtt{RCAS} \cup \mathtt{RFAA} \cup \mathtt{nF}) \\
& \mid \mathtt{nlR}\langle r, w, a, w'\rangle & \text{where } r \in \mathtt{nlR}, w \in \mathcal{W}, a \in \mathtt{Put}, w' \in \mathtt{nrW}, \mathsf{eq}_{\mathtt{loc\&v}}(r, w), \\
& & \quad \mathtt{loc}_r(a) = \mathtt{loc}(r), \mathtt{loc}_w(a) = \mathtt{loc}(w'), v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w') \\
& \mid \mathtt{nrR}\langle r, w, a, w'\rangle & \text{where } r \in \mathtt{nrR}, w \in \mathcal{W}, a \in \mathtt{Get}, w' \in \mathtt{nlW}, \mathsf{eq}_{\mathtt{loc\&v}}(r, w), \\
& & \quad \mathtt{loc}_r(a) = \mathtt{loc}(r), \mathtt{loc}_w(a) = \mathtt{loc}(w'), v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w') \\
& \mid \mathtt{narR}\langle r, w, a, w', w''\rangle & \text{where } r \in \mathtt{narR}, w \in \mathcal{W}, a \in \mathtt{RRMW}, w' \in \mathtt{nlW}, w'' \in \mathtt{narW}, \\
& & \quad \mathsf{eq}_{\mathtt{loc\&v}}(r, w), \mathtt{loc}_r(a) = \mathtt{loc}(r) = \mathtt{loc}(w''), \\
& & \quad \mathtt{loc}_w(a) = \mathtt{loc}(w'), v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w'), \\
& & \quad a \in \mathtt{RCAS} \implies v_{\mathrm{r}}(r) = v_{\mathrm{e}}(a) \wedge v_{\mathrm{w}}(w'') = v_{\mathrm{u}}(a) \\
& & \quad a \in \mathtt{RFAA} \implies v_{\mathrm{w}}(w'') = v_{\mathrm{r}}(r) + v(a) \\
& \mid \mathtt{naF}\langle r, w, a, w'\rangle & \text{where } r \in \mathtt{narR}, w \in \mathcal{W}, a \in \mathtt{rRMW}, w' \in \mathtt{nlW}, \mathsf{eq}_{\mathtt{loc\&v}}(r, w), \\
& & \quad \mathtt{loc}_r(a) = \mathtt{loc}(r), \mathtt{loc}_w(a) = \mathtt{loc}(w'), \\
& & \quad v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w^l), v_{\mathrm{r}}(r) \neq v_{\mathrm{e}}(a) \\
& \mid \mathtt{nlW}\langle w, e\rangle & \text{where } w \in \mathtt{nlW}, e \in \mathtt{nlEX}, \mathsf{sameqp}(w, e) \\
& \mid \mathtt{nrW}\langle w, e\rangle & \text{where } w \in \mathtt{nrW}, e \in \mathtt{nrEX}, \mathsf{sameqp}(w, e) \\
& \mid \mathtt{narW}\langle w\rangle & \text{where } w \in \mathtt{narW} \\
& \mid \mathsf{CN}\langle e\rangle & \text{where } e \in \mathtt{nrEX} \\
& \mid \mathtt{P}\langle p, e\rangle & \text{where } p \in \mathtt{P}, e \in \mathtt{nEX}, \mathsf{sameqp}(p, e) \\
& \mid \mathtt{nF}\langle f\rangle & \text{where } f \in \mathtt{nF} \\
& \mid \mathsf{B}\langle w\rangle & \text{where } w \in \mathcal{W} \\
& \mid \mathcal{E}\langle t\rangle & \text{where } t \in \mathsf{Tid}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{eq}_{\mathtt{loc\&v}}(r, w) & \triangleq & \mathtt{loc}(r) = \mathtt{loc}(w) \wedge v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w) \\
\mathsf{sameqp}(e, e') & \triangleq & t(e) = t(e') \wedge \overline{n}(e) = \overline{n}(e')
\end{array}
$$

Fig. 24: Annotated Labels

Program transitions: $\mathsf{Prog} \xrightarrow{\mathsf{ALabel} \uplus \{\mathtt{CASF}\}} \mathsf{Prog}$
Command transitions: $\mathsf{Comm} \xrightarrow{\mathsf{ALabel} \uplus \{\mathtt{CASF}\}} \mathsf{Comm}$

$$\frac{\mathsf{C}_1 \xrightarrow{\lambda} \mathsf{C}_1'}{\mathsf{C}_1; \mathsf{C}_2 \xrightarrow{\lambda} \mathsf{C}_1'; \mathsf{C}_2} \qquad \frac{}{\mathtt{skip}; \mathsf{C} \xrightarrow{\mathcal{E}\langle t \rangle} \mathsf{C}} \qquad \frac{i \in \{1,2\}}{\mathsf{C}_1 + \mathsf{C}_2 \xrightarrow{\mathcal{E}\langle t \rangle} \mathsf{C}_i} \qquad \frac{}{\mathsf{C}^* \xrightarrow{\mathcal{E}\langle t \rangle} \mathtt{skip}}$$

$$\frac{}{\mathsf{C}^* \xrightarrow{\mathcal{E}\langle t \rangle} \mathsf{C}; \mathsf{C}^*} \qquad \frac{\mathsf{C} \rightsquigarrow \mathsf{C}'}{\mathsf{C} \xrightarrow{\mathcal{E}\langle t \rangle} \mathsf{C}'} \qquad \frac{\mathsf{elocs}(e) = \emptyset \qquad w = (\iota, t, \mathtt{lW}(x, [\![e]\!]))}{x := e \xrightarrow{\mathtt{lW}\langle w \rangle} \mathtt{skip}}$$

$$\frac{\mathsf{elocs}(e_{\mathrm{old}}) = \mathsf{elocs}(e_{\mathrm{new}}) = \emptyset \qquad v \neq [\![e_{\mathrm{old}}]\!] \qquad r = (\iota, t, \mathtt{lR}(x, v))}{z := \mathtt{CAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}) \xrightarrow{\mathtt{CASF}\langle r, w \rangle} z := v}$$

$$\frac{\mathsf{elocs}(e_{\mathrm{old}}) = \mathsf{elocs}(e_{\mathrm{new}}) = \emptyset \qquad u = (\iota, t, \mathtt{CAS}(x, [\![e_{\mathrm{old}}]\!], [\![e_{\mathrm{new}}]\!]))}{z := \mathtt{CAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}) \xrightarrow{\mathtt{CAS}\langle u, w \rangle} z := [\![e_{\mathrm{old}}]\!]}$$

$$\frac{f = (\iota, t, \mathtt{F})}{\mathtt{mfence} \xrightarrow{\mathtt{F}\langle f \rangle} \mathtt{skip}} \qquad \frac{a = (\iota, t, \mathtt{Get}(x, \overline{y}))}{x := \overline{y} \xrightarrow{\mathtt{Push}\langle a \rangle} \mathtt{skip}} \qquad \frac{a = (\iota, t, \mathtt{Put}(\overline{y}, x))}{\overline{y} := x \xrightarrow{\mathtt{Push}\langle a \rangle} \mathtt{skip}}$$

$$\frac{a = (\iota, t, \mathtt{nF}(\overline{n}))}{\mathtt{rfence}(\overline{n}) \xrightarrow{\mathtt{Push}\langle a \rangle} \mathtt{skip}} \qquad \frac{\begin{array}{c}\mathsf{elocs}(e_{\mathrm{old}}) = \mathsf{elocs}(e_{\mathrm{new}}) = \emptyset \\ v = [\![e_{\mathrm{old}}]\!] \qquad u = [\![e_{\mathrm{new}}]\!] \qquad a = (\iota, t, \mathtt{RCAS}(z, \overline{x}, v, u))\end{array}}{z := \mathtt{RCAS}(x, e_{\mathrm{old}}, e_{\mathrm{new}}) \xrightarrow{\mathtt{Push}\langle a \rangle} \mathtt{skip}}$$

$$\frac{\mathsf{elocs}(e) = \emptyset \qquad u = [\![e]\!] \qquad a = (\iota, t, \mathtt{RFAA}(z, \overline{x}, u))}{z := \mathtt{RFAA}(x, e) \xrightarrow{\mathtt{Push}\langle a \rangle} \mathtt{skip}} \qquad \frac{p = (\iota, t, \mathtt{P}(n))}{\mathtt{poll}(n) \xrightarrow{\mathtt{P}\langle p, e \rangle} \mathtt{skip}}$$

$$\frac{r = (\iota, t, \mathtt{lR}(x, v))}{\mathtt{assume}(x = v) \xrightarrow{\mathtt{lR}\langle r, w \rangle} \mathtt{skip}} \qquad \frac{v \neq v' \qquad r = (\iota, t, \mathtt{lR}(x, v'))}{\mathtt{assume}(x \neq v) \xrightarrow{\mathtt{lR}\langle r, w \rangle} \mathtt{skip}} \qquad \frac{\mathsf{P}(t(\lambda)) \xrightarrow{\lambda} \mathsf{C}}{\mathsf{P} \xrightarrow{\lambda} \mathsf{P}[t(\lambda) \mapsto \mathsf{C}]}$$

Fig. 25: RDMA$^{\mathrm{TSO}}$ program and command transitions for the annotated semantics

$$\mathsf{M} \in \mathsf{AMem} \triangleq \{m \in \mathsf{Loc} \to \mathcal{W} \mid \forall x \in \mathsf{Loc}.\mathtt{loc}(m[x]) = x\} \qquad \mathsf{B} \in \mathsf{ASBMap} \triangleq \mathsf{Tid} \to \mathsf{ASBuff}$$
$$\mathsf{A} \in \mathsf{RAMap} \triangleq \lambda n.\, \{\bot, \top\} \qquad \mathsf{QP} \in \mathsf{AQPMap} \triangleq \mathsf{Tid} \to (\mathsf{Node} \to \mathsf{AQPair})$$
$$\mathsf{b} \in \mathsf{ASBuff} \triangleq (\mathtt{lW} \cup \mathtt{Get} \cup \mathtt{Put} \cup \mathtt{nF} \cup \ \mathtt{RCAS} \cup \mathtt{RFAA})^* \qquad \mathbf{sqp} \in \mathsf{AQPair} \triangleq \mathsf{APipe} \times \mathsf{AWBR} \times \mathsf{AWBL}$$
$$\mathbf{pipe} \in \mathsf{APipe} \triangleq (\mathtt{Get} \cup \mathtt{Put} \cup \mathtt{nF} \cup \mathtt{nrW} \cup \ \mathtt{narW} \cup \mathtt{nrEX} \cup \mathtt{nlW} \cup \ \mathtt{RCAS} \cup \mathtt{RFAA})^*$$
$$\mathbf{wb_R} \in \mathsf{AWBR} \triangleq (\mathtt{nrW}, \mathtt{narW})^* \qquad \mathbf{wb_L} \in \mathsf{AWBL} \triangleq (\mathtt{nlW} \cup \mathtt{nlEX} \cup \mathtt{nrEX})^*$$

$$\frac{\mathsf{B}' = \mathsf{B}[t(w) \mapsto w \cdot \mathsf{B}(t(w))]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\mathtt{lW}\langle w\rangle} \mathsf{M},\mathsf{B}',\mathsf{A},\mathsf{QP}} \qquad \frac{(\mathsf{M} \blacktriangleleft \mathsf{B}(t(r)))(\mathtt{loc}(r)) = w \qquad v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w)}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\mathtt{lR}\langle r,w\rangle} \mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}}$$

$$\frac{\mathsf{B}(t(u)) = \varepsilon \qquad \mathsf{M}(\mathtt{loc}(u)) = w \qquad v_{\mathrm{r}}(u) = v_{\mathrm{w}}(w)}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\mathtt{CAS}\langle u,w\rangle} \mathsf{M}[x \mapsto u],\mathsf{B},\mathsf{A},\mathsf{QP}} \qquad \frac{\mathsf{B}(t(f)) = \varepsilon}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\mathtt{F}\langle f\rangle} \mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}}$$

$$\frac{\mathsf{B}' = \mathsf{B}[t(a) \mapsto a \cdot \mathsf{B}(t(a))]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\mathtt{Push}\langle a\rangle} \mathsf{M},\mathsf{B}',\mathsf{A},\mathsf{QP}} \qquad \frac{\mathsf{B}(t(w)) = \mathsf{b} \cdot w \qquad w \in \mathtt{lW}}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\mathtt{B}\langle w\rangle} \mathsf{M}[x \mapsto w],\mathsf{B}[t(w) \mapsto \mathsf{b}],\mathsf{A},\mathsf{QP}}$$

$$\frac{\mathsf{B}(t(a)) = \mathsf{b} \cdot a \qquad a \notin \mathtt{lW} \qquad \mathsf{QP}(t(a))(n(a)) = \mathsf{qp} \qquad \mathsf{qp}' = \mathsf{qp}[\mathbf{pipe} \mapsto a \cdot \mathsf{qp}.\mathbf{pipe}]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\mathtt{NIC}\langle a\rangle} \mathsf{M},\mathsf{B}[t(a) \mapsto \mathsf{b}],\mathsf{A},\mathsf{QP}[t(a) \mapsto \mathsf{QP}(t(a))[n(a) \mapsto \mathsf{qp}']]}$$

$$\frac{\mathsf{QP}(t(p))(n(p)) = \mathsf{qp} \qquad \mathsf{qp}.\mathbf{wb_L} = \alpha \cdot e \qquad e \in \mathtt{nEX} \qquad \mathsf{qp}' = \mathsf{qp}[\mathbf{wb_L} \mapsto \alpha]}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\mathtt{P}\langle p,e\rangle} \mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP}[t(p) \mapsto \mathsf{QP}(t(p))[n(p) \mapsto \mathsf{qp}']]}$$

$$\frac{\mathsf{M},\mathsf{A},\mathsf{QP}(t(\lambda))(\overline{n}) \xrightarrow{\lambda}_{\mathtt{sqp}} \mathsf{M}',\mathsf{A}',\mathsf{qp}}{\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP} \xrightarrow{\lambda} \mathsf{M}',\mathsf{B},\mathsf{A}',\mathsf{QP}[t(\lambda) \mapsto \mathsf{QP}(t(\lambda))[\overline{n} \mapsto \mathsf{qp}]]}$$

$$\text{with} \quad (\mathsf{M} \blacktriangleleft \alpha)(x) = \begin{cases} \mathsf{M}[x] & \alpha = \varepsilon \\ w & \alpha = w \cdot \beta \wedge w \in \mathcal{W} \wedge \mathtt{loc}(w) = x \\ (\mathsf{M} \blacktriangleleft \beta)(x) & \alpha = e \cdot \beta \wedge (e \notin \mathcal{W} \vee \mathtt{loc}(e) \neq x) \end{cases}$$

Fig. 26: RDMA$^{\mathrm{TSO}}$ hardware domains and hardware transitions for the annotated semantics

$$\frac{\mathbf{pipe} = \alpha \cdot f \qquad f = (\iota, t, \mathtt{nF}(n))}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{nF}\langle f \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \langle \alpha, \mathbf{wb_R}, \mathbf{wb_L} \rangle}$$

$$\frac{\begin{array}{c}\mathbf{pipe} = \alpha \cdot a \cdot \beta \qquad a = (\iota_a, t, \mathtt{Put}(\overline{y}, x)) \qquad \mathsf{M}(x) = w \qquad r = (\iota_r, t, \mathtt{nlR}(x, v_{\mathrm{w}}(w), n(\overline{y})))\\ w' = (\iota_{w'}, t, \mathtt{nrW}(\overline{y}, v_{\mathrm{w}}(w))) \qquad \beta \in (\mathtt{nrW} \cup \mathtt{narW} \cup \mathtt{Get} \cup \mathtt{nlW} \cup \mathtt{RCAS} \cup \mathtt{RFAA} \cup \mathtt{nrEX})^* \qquad \mathbf{wb_L} \in \mathtt{nEX}^*\end{array}}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{nlR}\langle r, w, a, w' \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \langle \alpha \cdot w' \cdot \beta, \mathbf{wb_R}, \mathbf{wb_L} \rangle}$$

$$\frac{\mathbf{pipe} = \alpha \cdot w \cdot \beta \qquad w = (\iota_w, t, \mathtt{nrW}(\overline{y}, v)) \qquad e = (\iota_e, t, \mathtt{nrEX}(n(\overline{y}))) \qquad \beta \in (\mathtt{Get} \cup \mathtt{nlW} \cup \mathtt{nrEX})^*}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{nrW}\langle w, e \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \langle \alpha \cdot e \cdot \beta, w \cdot \mathbf{wb_R}, \mathbf{wb_L} \rangle}$$

$$\frac{\mathbf{wb_R} = \alpha \cdot w \qquad w \in \mathtt{nrW}}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathsf{B}\langle w \rangle}_{\mathtt{sqp}} \mathsf{M}[\mathtt{loc}(w) \mapsto w], \mathsf{A}, \langle \mathbf{pipe}, \alpha, \mathbf{wb_L} \rangle}$$

$$\frac{\mathbf{pipe} = \alpha \cdot e \qquad e \in \mathtt{nrEX}}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathsf{CN}\langle e \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \langle \alpha, \mathbf{wb_R}, e \cdot \mathbf{wb_L} \rangle}$$

$$\frac{\begin{array}{c}\mathbf{pipe} = \alpha \cdot a \cdot \beta \qquad a = (\iota_a, t, \mathtt{Get}(x, \overline{y})) \qquad \mathsf{M}(\overline{y}) = w \qquad r = (\iota_r, t, \mathtt{nrR}(\overline{y}, v_{\mathrm{w}}(w)))\\ w' = (\iota_{w'}, t, \mathtt{nlW}(x, v_{\mathrm{w}}(w), n(\overline{y}))) \qquad \beta \in (\mathtt{Get} \cup \mathtt{nlW} \cup \mathtt{nrEX})^* \qquad \mathbf{wb_R} = \varepsilon\end{array}}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{nrR}\langle r, w, a, w' \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \langle \alpha \cdot w' \cdot \beta, \mathbf{wb_R}, \mathbf{wb_L} \rangle}$$

$$\frac{\mathbf{pipe} = \alpha \cdot w \qquad w = (\iota_w, t, \mathtt{nlW}(x, v, n)) \qquad e = (\iota_e, t, \mathtt{nlEX}(n))}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{nlW}\langle w, e \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \langle \alpha, \mathbf{wb_R}, e \cdot w \cdot \mathbf{wb_L} \rangle}$$

$$\frac{\mathbf{wb_L} = \alpha \cdot w \cdot \beta \qquad w \in \mathtt{nlW} \qquad \beta \in \mathtt{nEX}^*}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathsf{B}\langle w \rangle}_{\mathtt{sqp}} \mathsf{M}[\mathtt{loc}(w) \mapsto w], \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \alpha \cdot \beta \rangle}$$

$$\frac{\begin{array}{c}\mathbf{pipe} = \alpha \cdot a \cdot \beta \quad \mathbf{wb_R} = \varepsilon \quad \mathsf{M}(\overline{x}) = w\\ v_{\mathrm{w}}(w) \neq v \qquad \mathsf{A}(n(\overline{x})) = \bot \qquad a = (\iota_a, t, \mathtt{RCAS}(z, \overline{x}, v, u))\\ r = (\iota_r, t, \mathtt{narR}(\overline{x}, v_{\mathrm{w}}(w))) \qquad w' = (\iota_{w'}, t, \mathtt{nlW}(z, v_{\mathrm{w}}(w), n(\overline{x}))) \qquad \beta \in (\mathtt{Get} \cup \mathtt{nlW} \cup \mathtt{nrEX})^*\end{array}}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{naF}\langle r, w, a, w' \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \langle \alpha \cdot w' \cdot \beta, \mathbf{wb_R}, \mathbf{wb_L} \rangle}$$

$$\frac{\begin{array}{c}\mathbf{pipe} = \alpha \cdot a \cdot \beta \quad \mathbf{wb_R} = \varepsilon \quad \mathsf{M}(\overline{x}) = w\\ v_{\mathrm{w}}(w) = v \qquad \mathsf{A}(n(\overline{x})) = \bot \qquad a = (\iota_a, t, \mathtt{RCAS}(z, \overline{x}, v, u)) \qquad r = (\iota_r, t, \mathtt{narR}(\overline{x}, v_{\mathrm{w}}(w)))\\ w'' = (\iota_{w''}, t, \mathtt{narW}(\overline{x}, u)) \qquad w' = (\iota_{w'}, t, \mathtt{nlW}(z, v_{\mathrm{w}}(w), n(\overline{x}))) \qquad \beta \in (\mathtt{Get} \cup \mathtt{nlW} \cup \mathtt{nrEX})^*\end{array}}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{narR}\langle r, w, a, w', w'' \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}[n(\overline{x}) \mapsto \top], \langle \alpha \cdot w' \cdot w'' \cdot \beta, \mathbf{wb_R}, \mathbf{wb_L} \rangle}$$

$$\frac{\begin{array}{c}\mathbf{pipe} = \alpha \cdot a \cdot \beta \quad \mathbf{wb_R} = \varepsilon \quad \mathsf{M}(\overline{x}) = w\\ v_{\mathrm{w}}(w) + v = u \qquad \mathsf{A}(n(\overline{x})) = \bot \qquad a = (\iota_a, t, \mathtt{RFAA}(z, \overline{x}, v)) \qquad r = (\iota_r, t, \mathtt{narR}(\overline{x}, v_{\mathrm{w}}(w)))\\ w'' = (\iota_{w''}, t, \mathtt{narW}(\overline{x}, u))) \qquad w' = (\iota_{w'}, t, \mathtt{nlW}(z, v_{\mathrm{w}}(w))) \qquad \beta \in (\mathtt{Get} \cup \mathtt{nlW} \cup \mathtt{nrEX})^*\end{array}}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{narR}\langle r, w, a, w', w'' \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}[n(\overline{x}) \mapsto \top], \langle \alpha \cdot w' \cdot w'' \cdot \beta, \mathbf{wb_R}, \mathbf{wb_L} \rangle}$$

$$\frac{\mathbf{pipe} = \alpha \cdot w \cdot \beta \qquad \beta \in (\mathtt{Get} \cup \mathtt{nlW} \cup \mathtt{nrEX})^* \qquad w = (\iota_w, t, \mathtt{narW}(\overline{x}, v))}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathtt{narW}\langle w \rangle}_{\mathtt{sqp}} \mathsf{M}, \mathsf{A}, \langle \alpha \cdot \beta, w \cdot \mathbf{wb_R}, \mathbf{wb_L} \rangle}$$

$$\frac{\mathbf{wb_R} = \alpha \cdot w \qquad w = (\iota_w, t, \mathtt{narW}(\overline{x}, v))}{\mathsf{M}, \mathsf{A}, \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \xrightarrow{\mathsf{B}\langle w \rangle}_{\mathtt{sqp}} \mathsf{M}[\mathtt{loc}(w) \mapsto w], \mathsf{A}[n(\overline{x}) \mapsto \bot], \langle \mathbf{pipe}, \alpha, \mathbf{wb_L} \rangle}$$

Fig. 27: Annotated 3 Buffers NIC Semantics

D.4 Paths, Gluing, and Other Definitions

We define a path as: $\pi \in \mathsf{Path} \triangleq (\mathsf{ALabel} \setminus \mathcal{E}\langle t \rangle)^*$

We define Annotated Operational Semantics Gluing with the following rules.

$$\frac{\mathsf{P} \xrightarrow{\mathcal{E}\langle t \rangle} \mathsf{P}'}{\mathsf{P}, \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi \Rightarrow \mathsf{P}', \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi}$$

$$\frac{\mathsf{P} \xrightarrow{\lambda} \mathsf{P}' \qquad \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP} \xrightarrow{\lambda} \mathsf{M}', \mathsf{B}', \mathsf{A}, \mathsf{QP}' \qquad \lambda \in (\mathtt{lR} \cup \mathtt{lW} \cup \mathtt{CAS} \cup \mathtt{F} \cup \mathtt{Push} \cup \mathtt{P}) \qquad \mathsf{fresh}(\lambda, \pi)}{\mathsf{P}, \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi \Rightarrow \mathsf{P}', \mathsf{M}', \mathsf{B}', \mathsf{A}, \mathsf{QP}', \lambda \cdot \pi}$$

$$\frac{\mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP} \xrightarrow{\lambda} \mathsf{M}', \mathsf{B}', \mathsf{A}', \mathsf{QP}' \qquad \lambda \in (\mathtt{NIC} \cup \mathtt{nlR} \cup \mathtt{nrR} \cup \mathtt{nlW} \cup \mathtt{nrW} \cup \mathtt{naF} \cup \mathtt{narR} \cup \mathtt{narW} \cup \mathsf{CN} \cup \mathtt{nF} \cup \mathsf{B}) \qquad \mathsf{fresh}(\lambda, \pi)}{\mathsf{P}, \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi \Rightarrow \mathsf{P}, \mathsf{M}', \mathsf{B}', \mathsf{A}', \mathsf{QP}', \lambda \cdot \pi}$$

$$\frac{\mathsf{P} \xrightarrow{\mathtt{CASF}\langle r, w \rangle} \mathsf{P}' \qquad \lambda_1 = \mathtt{F}\langle (\iota, t(r), \mathtt{F}) \rangle \qquad \lambda_2 = \mathtt{lR}\langle r, w \rangle \qquad \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP} \xrightarrow{\lambda_1} \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP} \xrightarrow{\lambda_2} \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP} \qquad \mathsf{fresh}(\lambda_1, \pi) \qquad \mathsf{fresh}(\lambda_2, \pi)}{\mathsf{P}, \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi \Rightarrow \mathsf{P}', \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \lambda_2 \cdot \lambda_1 \cdot \pi}$$

Two annotated labels are non-conflicting ($\lambda_1 \bowtie \lambda_2$) if they are of a different type or if their relevant arguments are disjoints. An annotated label is fresh if it does not conflict with any previous annotated label.

$$\mathbf{Relevant} : \mathsf{ALabel} \to 2^{E^{\text{ext}}}$$

$$\begin{aligned}
\mathbf{Relevant}(\mathtt{lR}\langle r, _ \rangle) &\triangleq \{r\} & \mathbf{Relevant}(\mathtt{narR}\langle r, _, a, w'', w' \rangle) &\triangleq \{r, a, w', w''\} \\
\mathbf{Relevant}(\mathtt{lW}\langle w \rangle) &\triangleq \{w\} & \mathbf{Relevant}(\mathtt{narW}\langle w \rangle) &\triangleq \{w\} \\
\mathbf{Relevant}(\mathtt{CAS}\langle u, _ \rangle) &\triangleq \{u\} & \mathbf{Relevant}(\mathtt{nlW}\langle w, e \rangle) &\triangleq \{w, e\} \\
\mathbf{Relevant}(\mathtt{F}\langle f \rangle) &\triangleq \{f\} & \mathbf{Relevant}(\mathtt{nrW}\langle w, e \rangle) &\triangleq \{w, e\} \\
\mathbf{Relevant}(\mathtt{Push}\langle a \rangle) &\triangleq \{a\} & \mathbf{Relevant}(\mathsf{CN}\langle e \rangle) &\triangleq \{e\} \\
\mathbf{Relevant}(\mathtt{NIC}\langle a \rangle) &\triangleq \{a\} & \mathbf{Relevant}(\mathtt{P}\langle p, e \rangle) &\triangleq \{p, e\} \\
\mathbf{Relevant}(\mathtt{nlR}\langle r, _, a, w' \rangle) &\triangleq \{r, a, w'\} & \mathbf{Relevant}(\mathtt{nF}\langle f \rangle) &\triangleq \{f\} \\
\mathbf{Relevant}(\mathtt{nrR}\langle r, _, a, w' \rangle) &\triangleq \{r, a, w'\} & \mathbf{Relevant}(\mathsf{B}\langle w \rangle) &\triangleq \{w\} \\
\mathbf{Relevant}(\mathtt{naF}\langle r, _, a, w' \rangle) &\triangleq \{r, a, w'\} & \mathbf{Relevant}(\mathcal{E}\langle _ \rangle) &\triangleq \{\}
\end{aligned}$$

$$\lambda_1 \bowtie \lambda_2 \triangleq \mathsf{type}(\lambda_1) \neq \mathsf{type}(\lambda_2) \vee \mathbf{Relevant}(\lambda_1) \cap \mathbf{Relevant}(\lambda_2) = \emptyset$$

$$\mathsf{fresh}(\lambda, \pi) \triangleq \forall \lambda' \in \pi,\ \lambda \bowtie \lambda'$$
$$\mathsf{nodup}(\pi) \triangleq \forall \pi_2, \lambda, \pi_1.\ \pi = \pi_2 \cdot \lambda \cdot \pi_1 \implies \mathsf{fresh}(\lambda, \pi_1)$$

Relevant(λ) are the arguments that are important to consider to avoid duplicating events. The excluded events are the write operations we lookup when reading. For instance:

- Having both $\texttt{lR}\langle r_1, w\rangle$ and $\texttt{lR}\langle r_2, w\rangle$ during an execution is fine, since w can be looked up any number of time.
- Having both $\texttt{nlR}\langle r_1, w_1, a, e_1\rangle$ and $\texttt{nlR}\langle r_2, w_2, a, e_2\rangle$ during an execution is problematic, since it means the put operation a is being run twice.

Completeness.

$$\begin{aligned}
\mathsf{complete}(\pi) \triangleq \forall a, w', e, r, w, f, w''.\ & \\
& \texttt{lW}\langle w\rangle \in \pi \implies \mathsf{B}\langle w\rangle \in \pi \\
\wedge\ & \texttt{Push}\langle a\rangle \in \pi \implies \texttt{NIC}\langle a\rangle \in \pi \\
\wedge\ & \texttt{NIC}\langle f\rangle \in \pi \wedge f \in \texttt{nF} \implies \texttt{nF}\langle f\rangle \in \pi \\
\wedge\ & \texttt{NIC}\langle a\rangle \in \pi \wedge a \in \texttt{Put} \implies \exists r, w, w'.\ \texttt{nlR}\langle r, w, a, w'\rangle \in \pi \\
\wedge\ & \texttt{NIC}\langle a\rangle \in \pi \wedge a \in \texttt{Get} \implies \exists r, w, w'.\ \texttt{nrR}\langle r, w, a, w'\rangle \in \pi \\
\wedge\ & \texttt{NIC}\langle a\rangle \in \pi \wedge a \in \texttt{RFAA} \implies \exists r, w, w'.\ \texttt{narR}\langle r, w, a, w', w''\rangle \in \pi \\
\wedge\ & \texttt{NIC}\langle a\rangle \in \pi \wedge a \in \texttt{RCAS} \implies \begin{pmatrix} \exists r, w, w'.\ \texttt{naF}\langle r, w, a, w'\rangle \in \pi \\ \vee\ \exists r, w, w', w''.\ \texttt{narR}\langle r, w, a, w', w''\rangle \in \pi \end{pmatrix} \\
\wedge\ & \texttt{nlR}\langle r, w, a, w'\rangle \in \pi \implies \exists e.\ \texttt{nrW}\langle w', e\rangle \in \pi \\
\wedge\ & \texttt{nrR}\langle r, w, a, w'\rangle \in \pi \implies \exists e.\ \texttt{nlW}\langle w', e\rangle \in \pi \\
\wedge\ & \texttt{narR}\langle r, w, a, w', w''\rangle \in \pi \implies \texttt{narW}\langle w''\rangle \in \pi \\
\wedge\ & \texttt{narR}\langle r, w, a, w', w''\rangle \in \pi \implies \exists e.\ \texttt{nlW}\langle w', e\rangle \in \pi \\
\wedge\ & \texttt{naF}\langle r, w, a, w'\rangle \in \pi \implies \exists e.\ \texttt{nlW}\langle w', e\rangle \in \pi \\
\wedge\ & \texttt{nlW}\langle w, e\rangle \in \pi \implies \mathsf{B}\langle w\rangle \in \pi \\
\wedge\ & \texttt{nrW}\langle w, e\rangle \in \pi \implies \mathsf{B}\langle w\rangle \in \pi \wedge \mathsf{CN}\langle e\rangle \in \pi \\
\wedge\ & \texttt{narW}\langle w\rangle \in \pi \implies \mathsf{B}\langle w\rangle \in \pi
\end{aligned}$$

Informal: every pending operation is done and (most) buffers are empty. Note that some `nEX` (i.e., completion notifications) might still be in $\mathbf{wb_L}$.

For a path π without duplicate (e.g. if $\mathsf{nodup}(\pi)$ holds), we define the total ordering of its annotated labels as follows. Note that the early part of the path is on the right.

$$\lambda_1 \prec_\pi \lambda_2 \triangleq \exists \pi_1, \pi_2, \pi_3 \text{ s.t. } \pi = \pi_3 \cdot \lambda_2 \cdot \pi_2 \cdot \lambda_1 \cdot \pi_1$$

Backward Completeness. (with ordering)

$$\mathsf{backComp}(\pi) \triangleq \forall a, w', e, r, w, f, p, w''.$$

$$\mathsf{B}\langle w\rangle \in \pi \implies \begin{pmatrix} \mathtt{lW}\langle w\rangle \prec_\pi \mathsf{B}\langle w\rangle \\ \vee\ \exists e.\mathtt{nlW}\langle w, e\rangle \prec_\pi \mathsf{B}\langle w\rangle \\ \vee\ \exists e.\mathtt{nrW}\langle w, e\rangle \prec_\pi \mathsf{B}\langle w\rangle \\ \vee\ \exists e.\mathtt{narW}\langle w\rangle \prec_\pi \mathsf{B}\langle w\rangle \end{pmatrix}$$

$$\wedge\ \mathtt{NIC}\langle a\rangle \in \pi \implies \mathtt{Push}\langle a\rangle \prec_\pi \mathtt{NIC}\langle a\rangle$$

$$\wedge\ \mathtt{nF}\langle f\rangle \in \pi \implies \mathtt{NIC}\langle f\rangle \prec_\pi \mathtt{nF}\langle f\rangle$$

$$\wedge\ \mathtt{nlR}\langle r, w, a, w'\rangle \in \pi \implies \mathtt{NIC}\langle a\rangle \prec_\pi \mathtt{nlR}\langle r, w, a, w'\rangle$$

$$\wedge\ \mathtt{nrR}\langle r, w, a, w'\rangle \in \pi \implies \mathtt{NIC}\langle a\rangle \prec_\pi \mathtt{nrR}\langle r, w, a, w'\rangle$$

$$\wedge\ \mathtt{naF}\langle r, w, a, w'\rangle \in \pi \implies \mathtt{NIC}\langle a\rangle \prec_\pi \mathtt{naF}\langle r, w, a, w'\rangle$$

$$\wedge\ \mathtt{narR}\langle r, w, a, w', w''\rangle \in \pi \implies \mathtt{NIC}\langle a\rangle \prec_\pi \mathtt{narR}\langle r, w, a, w', w''\rangle$$

$$\wedge\ \mathtt{nrW}\langle w', e\rangle \in \pi \implies \exists r, w, a.\ \mathtt{nlR}\langle r, w, a, w'\rangle \prec_\pi \mathtt{nrW}\langle w', e\rangle$$

$$\wedge\ \mathtt{nlW}\langle w', e\rangle \in \pi \implies \begin{pmatrix} \exists r, w, a.\ \mathtt{nrR}\langle r, w, a, w'\rangle \prec_\pi \mathtt{nlW}\langle w', e\rangle \\ \vee\ \exists r, w, a.\ \mathtt{naF}\langle r, w, a, w'\rangle \prec_\pi \mathtt{nlW}\langle w', e\rangle \\ \vee\ \exists r, w, a, w''.\ \begin{pmatrix} \mathtt{narR}\langle r, w, a, w', w''\rangle \\ \prec_\pi \mathtt{narW}\langle w''\rangle \prec_\pi \mathtt{nlW}\langle w', e\rangle \end{pmatrix} \end{pmatrix}$$

$$\wedge\ \mathtt{narW}\langle w'\rangle \in \pi \implies \exists r, w, a, w''.\mathtt{narR}\langle r, w, a, w', w''\rangle \prec_\pi \mathtt{narW}\langle w'\rangle$$

$$\wedge\ \mathsf{CN}\langle e\rangle \in \pi \implies \exists w.\ \mathtt{nrW}\langle w, e\rangle \prec_\pi \mathsf{CN}\langle e\rangle$$

$$\wedge\ \mathtt{P}\langle p, e\rangle \in \pi \implies \begin{pmatrix} \exists w.\ \mathtt{nlW}\langle w, e\rangle \prec_\pi \mathsf{B}\langle w\rangle \prec_\pi \mathtt{P}\langle p, e\rangle \\ \vee\ \mathsf{CN}\langle e\rangle \prec_\pi \mathtt{P}\langle p, e\rangle \end{pmatrix}$$

Poll Order.

$$\mathsf{pollOrder}(\pi) \triangleq \forall e_1, e_2.\ \begin{pmatrix} \mathsf{sameqp}(e_1, e_2) \\ \wedge\ \lambda_1 \in \{\mathtt{nlW}\langle \_, e_1\rangle, \mathsf{CN}\langle e_1\rangle\} \\ \wedge\ \lambda_2 \in \{\mathtt{nlW}\langle \_, e_2\rangle, \mathsf{CN}\langle e_2\rangle\} \\ \wedge\ \lambda_1 \prec_\pi \lambda_2 \\ \wedge\ \mathtt{P}\langle \_, e_2\rangle \in \pi \end{pmatrix} \implies \mathtt{P}\langle \_, e_1\rangle \prec_\pi \mathtt{P}\langle \_, e_2\rangle$$

Flush Order.

$$
\begin{aligned}
&\mathsf{bufFlushOrd}(\pi) \triangleq \\
&\quad \forall w_1, w_2 \in \mathtt{lW}.\ \begin{pmatrix} t(w_1) = t(w_2) \implies \\ (\mathsf{B}\langle w_2\rangle \in \pi \wedge \mathtt{lW}\langle w_1\rangle \prec_\pi \mathtt{lW}\langle w_2\rangle) \iff \mathsf{B}\langle w_1\rangle \prec_\pi \mathsf{B}\langle w_2\rangle \end{pmatrix} \\
\wedge\ &\forall a_1, a_2 \in (\mathtt{Get} \cup \mathtt{Put} \cup \mathtt{nF} \cup \mathtt{RCAS} \cup \mathtt{RFAA}). \\
&\quad \begin{pmatrix} t(a_1) = t(a_2) \implies \\ (\mathtt{NIC}\langle a_2\rangle \in \pi \wedge \mathtt{Push}\langle a_1\rangle \prec_\pi \mathtt{Push}\langle a_2\rangle) \iff \mathtt{NIC}\langle a_1\rangle \prec_\pi \mathtt{NIC}\langle a_2\rangle \end{pmatrix} \\
\wedge\ &\forall a_1 \in (\mathtt{Get} \cup \mathtt{Put} \cup \mathtt{nF} \cup \mathtt{RCAS} \cup \mathtt{RFAA}), w_2 \in \mathtt{lW}. \\
&\quad \begin{pmatrix} t(a_1) = t(w_2) \implies \\ \quad (\mathsf{B}\langle w_2\rangle \in \pi \wedge \mathtt{Push}\langle a_1\rangle \prec_\pi \mathtt{lW}\langle w_2\rangle) \iff \mathtt{NIC}\langle a_1\rangle \prec_\pi \mathsf{B}\langle w_2\rangle \\ \wedge (\mathtt{NIC}\langle a_1\rangle \in \pi \wedge \mathtt{lW}\langle w_2\rangle \prec_\pi \mathtt{Push}\langle a_1\rangle) \iff \mathsf{B}\langle w_2\rangle \prec_\pi \mathtt{NIC}\langle a_1\rangle \end{pmatrix} \\
\wedge\ &\forall w_1, w_2 \in \mathtt{nlW}.\ \begin{pmatrix} \mathsf{sameqp}(w_1, w_2) \implies \\ (\mathsf{B}\langle w_2\rangle \in \pi \wedge \mathtt{nlW}\langle w_1\rangle \prec_\pi \mathtt{nlW}\langle w_2\rangle) \iff \mathsf{B}\langle w_1\rangle \prec_\pi \mathsf{B}\langle w_2\rangle \end{pmatrix} \\
\wedge\ &\forall w_1, w_2 \in \mathtt{nrW}.\ \begin{pmatrix} \mathsf{sameqp}(w_1, w_2) \implies \\ (\mathsf{B}\langle w_2\rangle \in \pi \wedge \mathtt{nrW}\langle w_1\rangle \prec_\pi \mathtt{nrW}\langle w_2\rangle) \iff \mathsf{B}\langle w_1\rangle \prec_\pi \mathsf{B}\langle w_2\rangle \end{pmatrix} \\
\wedge\ &\forall w_1, w_2 \in \mathtt{narW}.\ \begin{pmatrix} \mathsf{sameqp}(w_1, w_2) \implies \\ (\mathsf{B}\langle w_2\rangle \in \pi \wedge \mathtt{narW}\langle w_1\rangle \prec_\pi \mathtt{narW}\langle w_2\rangle) \iff \mathsf{B}\langle w_1\rangle \prec_\pi \mathsf{B}\langle w_2\rangle \end{pmatrix} \\
\wedge\ &\forall w_1 \in \mathtt{nrW}, w_2 \in \mathtt{narW}.\ \begin{pmatrix} \mathsf{sameqp}(w_1, w_2) \implies \\ (\mathsf{B}\langle w_2\rangle \in \pi \wedge \mathtt{nrW}\langle w_1\rangle \prec_\pi \mathtt{narW}\langle w_2\rangle) \iff \mathsf{B}\langle w_1\rangle \prec_\pi \mathsf{B}\langle w_2\rangle \end{pmatrix} \\
\wedge\ &\forall w_1 \in \mathtt{narW}, w_2 \in \mathtt{nrW}.\ \begin{pmatrix} \mathsf{sameqp}(w_1, w_2) \implies \\ (\mathsf{B}\langle w_2\rangle \in \pi \wedge \mathtt{narW}\langle w_1\rangle \prec_\pi \mathtt{nrW}\langle w_2\rangle) \iff \mathsf{B}\langle w_1\rangle \prec_\pi \mathsf{B}\langle w_2\rangle \end{pmatrix} \\
\wedge\ &\forall w \in \mathtt{lW}, f \in \mathtt{F}.\ \mathtt{lW}\langle w\rangle \prec_\pi \mathtt{F}\langle f\rangle \wedge t(w) = t(f) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{F}\langle f\rangle \\
\wedge\ &\forall w \in \mathtt{lW}, u \in \mathtt{CAS}.\ \mathtt{lW}\langle w\rangle \prec_\pi \mathtt{CAS}\langle u, \_\rangle \wedge t(w) = t(u) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{CAS}\langle u, \_\rangle \\
\wedge\ &\forall w \in \mathtt{nlW}, r \in \mathtt{nlR}. \\
&\quad (\mathtt{nlW}\langle w, \_\rangle \prec_\pi \mathtt{nlR}\langle r, \_, \_, \_\rangle \wedge \mathsf{sameqp}(w, r)) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{nlR}\langle r, \_, \_, \_\rangle \\
\wedge\ &\forall w \in \mathtt{nrW}, r \in \mathtt{nrR}. \\
&\quad (\mathtt{nrW}\langle w, \_\rangle \prec_\pi \mathtt{nrR}\langle r, \_, \_, \_\rangle \wedge \mathsf{sameqp}(w, r)) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{nrR}\langle r, \_, \_, \_\rangle \\
\wedge\ &\forall w \in \mathtt{nrW}, r \in \mathtt{narR}. \\
&\quad (\mathtt{nrW}\langle w, \_\rangle \prec_\pi \mathtt{naF}\langle r, \_, \_, \_\rangle \wedge \mathsf{sameqp}(w, r)) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{naF}\langle r, \_, \_, \_\rangle \\
\wedge\ &\forall w \in \mathtt{nrW}, r \in \mathtt{narR}. \\
&\quad (\mathtt{nrW}\langle w, \_\rangle \prec_\pi \mathtt{narR}\langle r, \_, \_, \_, \_\rangle \wedge \mathsf{sameqp}(w, r)) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{narR}\langle r, \_, \_, \_, \_\rangle \\
\wedge\ &\forall w \in \mathtt{narW}, r \in \mathtt{nrR}. \\
&\quad (\mathtt{narW}\langle w\rangle \prec_\pi \mathtt{nrR}\langle r, \_, \_, \_\rangle \wedge \mathsf{sameqp}(w, r)) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{nrR}\langle r, \_, \_, \_\rangle \\
\wedge\ &\forall w \in \mathtt{narW}, r \in \mathtt{narR}. \\
&\quad (\mathtt{narW}\langle w\rangle \prec_\pi \mathtt{naF}\langle r, \_, \_, \_\rangle \wedge \mathsf{sameqp}(w, r)) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{naF}\langle r, \_, \_, \_\rangle \\
\wedge\ &\forall w \in \mathtt{narW}, r \in \mathtt{narR}. \\
&\quad (\mathtt{narW}\langle w\rangle \prec_\pi \mathtt{narR}\langle r, \_, \_, \_, \_\rangle \wedge \mathsf{sameqp}(w, r)) \implies \mathsf{B}\langle w\rangle \prec_\pi \mathtt{narR}\langle r, \_, \_, \_, \_\rangle
\end{aligned}
$$

NIC Order.

$$
\begin{aligned}
&\mathsf{nicActOrder}(\pi) \triangleq \forall a_1, a_2.\ \mathtt{NIC}\langle a_1\rangle \prec_\pi \mathtt{NIC}\langle a_2\rangle \wedge \mathsf{sameqp}(a_1, a_2) \implies \\
&\quad (a_1 \in \mathtt{nF} \wedge a_2 \in \mathtt{Get} \wedge \mathtt{nrR}\langle \_, \_, a_2, \_\rangle \in \pi \implies \mathtt{nF}\langle a_1\rangle \prec_\pi \mathtt{nrR}\langle \_, \_, a_2, \_\rangle) \\
&\wedge\ (a_1 \in \mathtt{nF} \wedge a_2 \in \mathtt{Put} \wedge \mathtt{nlR}\langle \_, \_, a_2, \_\rangle \in \pi \implies \mathtt{nF}\langle a_1\rangle \prec_\pi \mathtt{nlR}\langle \_, \_, a_2, \_\rangle) \\
&\wedge\ (a_1 \in \mathtt{nF} \wedge a_2 \in \mathtt{RCAS} \wedge \mathtt{naF}\langle \_, \_, a_2, \_\rangle \in \pi \implies \mathtt{nF}\langle a_1\rangle \prec_\pi \mathtt{naF}\langle \_, \_, a_2, \_,\rangle) \\
&\wedge\ (a_1 \in \mathtt{nF} \wedge a_2 \in \mathtt{rRMW} \wedge \mathtt{narR}\langle \_, \_, a_2, \_, \_\rangle \in \pi \implies \mathtt{nF}\langle a_1\rangle \prec_\pi \mathtt{narR}\langle \_, \_, a_2, \_, \_\rangle) \\
&\wedge\ (a_1 \in \mathtt{nF} \wedge a_2 \in \mathtt{nF} \wedge \mathtt{nF}\langle a_2\rangle \in \pi \implies \mathtt{nF}\langle a_1\rangle \prec_\pi \mathtt{nF}\langle a_2\rangle) \\
&\wedge\ (a_1 \in \mathtt{Get} \wedge a_2 \in \mathtt{nF} \wedge \mathtt{nF}\langle a_2\rangle \in \pi \implies \mathtt{nrR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nlW}\langle w_1, \_\rangle \prec_\pi \mathtt{nF}\langle a_2\rangle) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{nF} \wedge \mathtt{nF}\langle a_2\rangle \in \pi \\ \implies \mathtt{nlR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nrW}\langle w_1, e_1\rangle \prec_\pi \mathsf{CN}\langle e_1\rangle \prec_\pi \mathtt{nF}\langle a_2\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{RCAS} \wedge a_2 \in \mathtt{nF} \wedge \mathtt{nF}\langle a_2\rangle \in \pi \\ \implies \mathtt{narR}\langle \_, \_, a_1, w_1, w_2\rangle \prec_\pi \mathtt{narW}\langle w_2\rangle \prec_\pi \mathtt{nlW}\langle w_1, \_\rangle \prec_\pi \mathtt{nF}\langle a_2\rangle \\ \quad \vee\ \mathtt{naF}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nlW}\langle w_1, \_\rangle \prec_\pi \mathtt{nF}\langle a_2\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{RFAA} \wedge a_2 \in \mathtt{nF} \wedge \mathtt{nF}\langle a_2\rangle \in \pi \\ \implies \mathtt{narR}\langle \_, \_, a_1, w_1, w_2\rangle \prec_\pi \mathtt{narW}\langle w_2\rangle \prec_\pi \mathtt{nlW}\langle w_1, \_\rangle \prec_\pi \mathtt{nF}\langle a_2\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Get} \wedge a_2 \in \mathtt{Get} \wedge \mathtt{nrR}\langle \_, \_, a_2, w_2\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \\ \implies \mathtt{nrR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nlW}\langle w_1, \_\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Get} \wedge a_2 \in \mathtt{Put} \wedge \mathtt{nlR}\langle \_, \_, a_2, w_2\rangle \prec_\pi \mathtt{nrW}\langle w_2, e_2\rangle \prec_\pi \mathsf{CN}\langle e_2\rangle \\ \implies \mathtt{nrR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nlW}\langle w_1, \_\rangle \prec_\pi \mathsf{CN}\langle e_2\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Get} \wedge a_2 \in \mathtt{RCAS} \wedge \left(\begin{array}{l} \mathtt{narR}\langle \_, \_, a_2, w_2, \_\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \\ \vee\ \mathtt{naF}\langle \_, \_, a_2, w_2\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \end{array}\right) \\ \implies \mathtt{nrR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nlW}\langle w_1, \_\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Get} \wedge a_2 \in \mathtt{RFAA} \wedge \mathtt{narR}\langle \_, \_, a_2, w_2, \_\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \\ \implies \mathtt{nrR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nlW}\langle w_1, \_\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{Get} \wedge \mathtt{nrR}\langle \_, \_, a_2, \_\rangle \in \pi \\ \implies \mathtt{nlR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nrW}\langle w_1, \_\rangle \prec_\pi \mathtt{nrR}\langle \_, \_, a_2, \_\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{Get} \wedge \mathtt{nrR}\langle \_, \_, a_2, w_2\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \\ \implies \mathtt{nlR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nrW}\langle w_1, e_1\rangle \prec_\pi \mathsf{CN}\langle e_1\rangle \prec_\pi \mathtt{nlW}\langle w_2, \_\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{RCAS} \wedge \mathtt{naF}\langle \_, \_, a_2, w_2\rangle \in \pi \\ \implies \mathtt{nlR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nrW}\langle w_1, e_1\rangle \prec_\pi \mathtt{naF}\langle \_, \_, a_2, w_2\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{RCAS} \cup \mathtt{RFAA} \wedge \mathtt{narR}\langle \_, \_, a_2, w_2, \_\rangle \in \pi \\ \implies \mathtt{nlR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nrW}\langle w_1, e_1\rangle \prec_\pi \mathtt{narR}\langle \_, \_, a_2, w_2, \_\rangle \end{array}\right) \\
&\wedge\ (a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{Put} \wedge \mathtt{nlR}\langle \_, \_, a_2, \_\rangle \in \pi \implies \mathtt{nlR}\langle \_, \_, a_1, \_\rangle \prec_\pi \mathtt{nlR}\langle \_, \_, a_2, \_\rangle) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{Put} \wedge \mathtt{nlR}\langle \_, \_, a_2, w_2\rangle \prec_\pi \mathtt{nrW}\langle w_2, \_\rangle \\ \implies \mathtt{nlR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nrW}\langle w_1, \_\rangle \prec_\pi \mathtt{nrW}\langle w_2, \_\rangle \end{array}\right) \\
&\wedge\ \left(\begin{array}{l} a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{Put} \wedge \mathtt{nlR}\langle \_, \_, a_2, w_2\rangle \prec_\pi \mathtt{nrW}\langle w_2, e_2\rangle \prec_\pi \mathsf{CN}\langle e_2\rangle \\ \implies \mathtt{nlR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nrW}\langle w_1, e_1\rangle \prec_\pi \mathsf{CN}\langle e_1\rangle \prec_\pi \mathsf{CN}\langle e_2\rangle \end{array}\right)
\end{aligned}
$$

$$
\begin{array}{l}
\wedge \left(\begin{array}{l} a_1 \in (\texttt{RCAS} \cup \texttt{RFAA}) \wedge a_2 \in \texttt{Get} \wedge \texttt{nrR}\langle \_, \_, a_2, \_\rangle \in \pi \\ \implies \left(\begin{array}{l} a_1 \in \texttt{RCAS} \wedge \texttt{naF}\langle \_, \_, a_1, \_\rangle \prec_\pi \texttt{nrR}\langle \_, \_, a_2, w_2\rangle \\ \vee\ \texttt{narR}\langle \_, \_, a_1, w_1, \_\rangle \prec_\pi \texttt{narW}\langle w_1\rangle \prec_\pi \texttt{nrR}\langle \_, \_, a_2, w_2\rangle \end{array} \right) \end{array} \right) \\
\wedge \left(\begin{array}{l} a_1 \in (\texttt{RCAS} \cup \texttt{RFAA}) \wedge a_2 \in \texttt{Get} \wedge \texttt{nrR}\langle \_, \_, a_2, w_2\rangle \prec_\pi \texttt{nlW}\langle w_2, \_\rangle \\ \implies \left(\begin{array}{l} a_1 \in \texttt{RCAS} \wedge \texttt{naF}\langle \_, \_, a_1, w_1\rangle \prec_\pi \texttt{nlW}\langle w_1, \_\rangle \prec_\pi \texttt{nlW}\langle w_2, \_\rangle \\ \vee\ \texttt{narR}\langle \_, \_, a_1, w_1, \_\rangle \prec_\pi \texttt{nlW}\langle w_1, \_\rangle \prec_\pi \texttt{nlW}\langle w_2, \_\rangle \end{array} \right) \end{array} \right) \\
\wedge \left(\begin{array}{l} a_1 \in (\texttt{RCAS} \cup \texttt{RFAA}) \wedge a_2 \in \texttt{Put} \wedge \texttt{nlR}\langle \_, \_, a_2, w_2\rangle \prec_\pi \texttt{nrW}\langle w_2, \_\rangle \\ \implies \left(\begin{array}{l} a_1 \in \texttt{RCAS} \wedge \texttt{naF}\langle \_, \_, a_1, w_1\rangle \prec_\pi \texttt{nrW}\langle w_2, \_\rangle \\ \vee\ \texttt{narR}\langle \_, \_, a_1, w_1, \_\rangle \prec_\pi \texttt{narW}\langle w_1\rangle \prec_\pi \texttt{nrW}\langle w_2, \_\rangle \end{array} \right) \end{array} \right) \\
\wedge \left(\begin{array}{l} a_1 \in (\texttt{RCAS} \cup \texttt{RFAA}) \wedge a_2 \in \texttt{Put} \wedge \texttt{nlR}\langle \_, \_, a_2, w_2\rangle \prec_\pi \texttt{nrW}\langle w_2, e_2\rangle \prec_\pi \mathsf{CN}\langle e_2\rangle \\ \implies \left(\begin{array}{l} a_1 \in \texttt{RCAS} \wedge \texttt{naF}\langle \_, \_, a_1, w_1\rangle \prec_\pi \texttt{nlW}\langle w_1, \_\rangle \prec_\pi \mathsf{CN}\langle e_2\rangle \\ \vee\ \texttt{narR}\langle \_, \_, a_1, w_1, \_\rangle \prec_\pi \texttt{nlW}\langle w_1, \_\rangle \prec_\pi \mathsf{CN}\langle e_2\rangle \end{array} \right) \end{array} \right) \\
\wedge \left(\begin{array}{l} a_1 \in (\texttt{RCAS} \cup \texttt{RFAA}) \wedge a_2 \in \texttt{RCAS} \wedge \texttt{naF}\langle \_, \_, a_2, \_\rangle \in \pi \\ \implies \left(\begin{array}{l} a_1 \in \texttt{RCAS} \wedge \texttt{naF}\langle \_, \_, a_1, \_\rangle \prec_\pi \texttt{naF}\langle \_, \_, a_2, \_\rangle \\ \vee\ \texttt{narR}\langle \_, \_, a_1, \_, w_1\rangle \prec_\pi \texttt{narW}\langle w_1\rangle \prec_\pi \texttt{naF}\langle \_, \_, a_2, \_\rangle \end{array} \right) \end{array} \right) \\
\wedge \left(\begin{array}{l} a_1, a_2 \in (\texttt{RCAS} \cup \texttt{RFAA}) \wedge \texttt{narR}\langle \_, \_, a_2, \_, \_\rangle \in \pi \\ \implies \left(\begin{array}{l} a_1 \in \texttt{RCAS} \wedge \texttt{naF}\langle \_, \_, a_1, w_1\rangle \prec_\pi \texttt{narR}\langle \_, \_, a_2, \_, \_\rangle \\ \vee\ \texttt{narR}\langle \_, \_, a_1, \_, w_1\rangle \prec_\pi \texttt{narW}\langle w_1\rangle \prec_\pi \texttt{narR}\langle \_, \_, a_2, \_, \_\rangle \end{array} \right) \end{array} \right) \\
\wedge \left(\begin{array}{l} a_1, a_2 \in (\texttt{RCAS} \cup \texttt{RFAA}) \wedge \left(\begin{array}{l} a_1 \in \texttt{RCAS} \wedge \texttt{naF}\langle \_, \_, a_2, w_2\rangle \prec_\pi \texttt{nlW}\langle w_2, \_\rangle \\ \vee\ \texttt{narR}\langle \_, \_, a_2, w_2, \_\rangle \prec_\pi \texttt{nlW}\langle w_2, \_\rangle \end{array} \right) \\ \implies \left(\begin{array}{l} a_1 \in \texttt{RCAS} \wedge \texttt{naF}\langle \_, \_, a_1, w_1\rangle \prec_\pi \texttt{nlW}\langle w_1, \_\rangle \prec_\pi \texttt{nlW}\langle w_2, \_\rangle \\ \vee\ \texttt{narR}\langle \_, \_, a_1, w_1, \_\rangle \prec_\pi \texttt{nlW}\langle w_1, \_\rangle \prec_\pi \texttt{nlW}\langle w_2, \_\rangle \end{array} \right) \end{array} \right)
\end{array}
$$

NIC Atomicity.

$$
\mathsf{nicAtomicity}(\pi) \triangleq \forall a_1, a_2, r, w.
\left(\begin{array}{l} \lambda_1 = \texttt{narR}\langle r_1, \_, a_1, \_, w\rangle \\ \wedge\ \lambda_2 \in \{\texttt{naF}\langle \_, \_, a_2, \_\rangle, \texttt{narR}\langle \_, \_, a_2, \_, \_\rangle\} \\ \wedge\ a_1, a_2 \in \texttt{rRMW} \ \wedge\ \overline{n}(a_1) = \overline{n}(a_2) \ \wedge\ \lambda_1 \prec_\pi \lambda_2 \end{array} \right) \implies \mathsf{B}\langle w\rangle \prec_\pi \lambda_2
$$

Read Order.

$$
\begin{array}{rl}
\mathsf{wfrd}(\pi) \triangleq & \forall \pi_2, r, w, \pi_1.\ \pi = \pi_2 \cdot \texttt{lR}\langle r, w\rangle \cdot \pi_1 \implies \mathsf{wfrdCPU}(r, w, \pi_1) \\
\wedge & \forall \pi_2, u, w, \pi_1.\ \pi = \pi_2 \cdot \texttt{CAS}\langle u, w\rangle \cdot \pi_1 \implies \mathsf{wfrdCPU}(u, w, \pi_1) \\
\wedge & \forall \pi_2, r, w, \pi_1.\ \pi = \pi_2 \cdot \texttt{nlR}\langle r, w, \_, \_\rangle \cdot \pi_1 \implies \mathsf{wfrdNIC}(r, w, \pi_1) \\
\wedge & \forall \pi_2, r, w, \pi_1.\ \pi = \pi_2 \cdot \texttt{nrR}\langle r, w, \_, \_\rangle \cdot \pi_1 \implies \mathsf{wfrdNIC}(r, w, \pi_1) \\
\wedge & \forall \pi_2, r, w, \pi_1.\ \pi = \pi_2 \cdot \texttt{naF}\langle r, w, \_, \_\rangle \cdot \pi_1 \implies \mathsf{wfrdNIC}(r, w, \pi_1) \\
\wedge & \forall \pi_2, r, w, \pi_1.\ \pi = \pi_2 \cdot \texttt{narR}\langle r, w, \_, \_, \_\rangle \cdot \pi_1 \implies \mathsf{wfrdNIC}(r, w, \pi_1)
\end{array}
$$

$$\mathsf{wfrdCPU}(r,w,\pi) \triangleq \begin{pmatrix} \exists \pi_2, \lambda, \pi_1.\ \pi = \pi_2 \cdot \lambda \cdot \pi_1 \\ \wedge\ \lambda \in \{\mathsf{B}\langle w\rangle, \mathtt{CAS}\langle w, \_\rangle\} \\ \wedge\ \{\mathsf{B}\langle w'\rangle, \mathtt{CAS}\langle w', \_\rangle \in \pi_2 \mid \mathtt{loc}(w') = \mathtt{loc}(r)\} = \emptyset \\ \wedge\ \left\{ w' \;\middle|\; \begin{array}{l} \mathtt{lW}\langle w'\rangle \in \pi \wedge \mathsf{B}\langle w'\rangle \notin \pi \wedge \\ \mathtt{loc}(w') = \mathtt{loc}(r) \wedge t(w') = t(r) \end{array} \right\} = \emptyset \end{pmatrix}$$
$$\vee \begin{pmatrix} \exists \pi_2, \lambda, \pi_1.\ \pi = \pi_2 \cdot \lambda \cdot \pi_1 \\ \wedge\ \lambda = \mathtt{lW}\langle w\rangle \wedge t(w) = t(r) \wedge \mathsf{B}\langle w\rangle \notin \pi_2 \\ \wedge\ \{\mathtt{lW}\langle w'\rangle \in \pi_2 \mid \mathtt{loc}(w') = \mathtt{loc}(r) \wedge t(w') = t(r)\} = \emptyset \end{pmatrix}$$
$$\vee \begin{pmatrix} w = init_{\mathtt{loc}(w)} \wedge \\ \left\{ \begin{array}{l} \mathsf{B}\langle w'\rangle, \mathtt{CAS}\langle w', \_\rangle \in \pi, \\ \mathtt{lW}\langle w''\rangle \in \pi \end{array} \;\middle|\; \begin{array}{l} \mathtt{loc}(w') = \mathtt{loc}(r) \wedge \\ \mathtt{loc}(w'') = \mathtt{loc}(r) \wedge t(w'') = t(r) \end{array} \right\} = \emptyset \end{pmatrix}$$

$$\mathsf{wfrdNIC}(r,w,\pi) \triangleq \begin{pmatrix} \exists \pi_2, \lambda, \pi_1.\ \pi = \pi_2 \cdot \lambda \cdot \pi_1 \\ \wedge\ \lambda \in \{\mathsf{B}\langle w\rangle, \mathtt{CAS}\langle w, \_\rangle\} \\ \wedge\ \{\mathsf{B}\langle w'\rangle, \mathtt{CAS}\langle w', \_\rangle \in \pi_2 \mid \mathtt{loc}(w') = \mathtt{loc}(r)\} = \emptyset \end{pmatrix}$$
$$\vee \begin{pmatrix} w = init_{\mathtt{loc}(w)} \wedge \\ \{\mathsf{B}\langle w'\rangle, \mathtt{CAS}\langle w', \_\rangle \in \pi \mid \mathtt{loc}(w') = \mathtt{loc}(r)\} = \emptyset \end{pmatrix}$$

Well-formed path.

$$\begin{aligned} \mathsf{wfp}(\pi) \triangleq\ & \mathsf{nodup}(\pi) \\ \wedge\ & \mathsf{backComp}(\pi) \\ \wedge\ & \mathsf{bufFlushOrd}(\pi) \\ \wedge\ & \mathsf{pollOrder}(\pi) \\ \wedge\ & \mathsf{nicActOrder}(\pi) \\ \wedge\ & \mathsf{nicAtomicity}(\pi) \\ \wedge\ & \mathsf{wfrd}(\pi) \end{aligned}$$

Definition 21.

$$\begin{aligned} \mathsf{wf}(\mathsf{M},\mathsf{B},\mathsf{A},\mathsf{QP},\pi) \triangleq\ & \mathsf{wfp}(\pi) \\ \wedge\ & \forall x \in \mathsf{Loc}.\ \mathsf{M}(x) = \mathsf{read}(\pi, x) \\ \wedge\ & \forall t \in \mathsf{Tid}.\mathsf{B}(t) = \mathsf{mksbuff}(\varepsilon, t, \pi) \\ \wedge\ & \forall n \in \mathsf{Node}.\mathsf{A}(n) = \mathsf{chkatm}(n, \pi) \\ \wedge\ & \forall t \in \mathsf{Tid}. \forall \overline{n} \in (\mathsf{Node} \setminus \{n(t)\}). \begin{pmatrix} \mathsf{QP}(t)(\overline{n}).\mathbf{pipe} = \mathsf{mkpipe}(\varepsilon, t, \overline{n}, \pi) \\ \mathsf{QP}(t)(\overline{n}).\mathbf{wb_R} = \mathsf{mkwbR}(\varepsilon, t, \overline{n}, \pi) \\ \mathsf{QP}(t)(\overline{n}).\mathbf{wb_L} = \mathsf{mkwbL}(\varepsilon, t, \overline{n}, \pi) \end{pmatrix} \end{aligned}$$

Where, the functions read, $\mathsf{mksbuff}$, chkatm, mkpipe, mkwbR, and mkwbL are defined below.

$$\mathsf{read}(\lambda{\cdot}\pi, x) \triangleq \begin{cases} w & \lambda \in \{\mathsf{B}\langle w\rangle, \mathtt{CAS}\langle w, \_\rangle\} \wedge \mathtt{loc}(w) = x \\ \mathsf{read}(\pi, x) & \text{otherwise} \end{cases}$$

$$\mathsf{read}(\varepsilon, x) \triangleq init_x$$

$$\mathsf{mksbuff}(\mathsf{b}, t, \varepsilon) \triangleq \mathsf{b}$$

$$\mathsf{mksbuff}(\mathsf{b}, t, \pi{\cdot}\lambda) \triangleq \begin{cases} \mathsf{mksbuff}(w{\cdot}\mathsf{b}, t, \pi) & \lambda = \mathtt{lW}\langle w\rangle \wedge t(w) = t \wedge \mathsf{B}\langle w\rangle \notin \pi \\ \mathsf{mksbuff}(a{\cdot}\mathsf{b}, t, \pi) & \lambda = \mathtt{Push}\langle a\rangle \wedge \mathtt{NIC}\langle a\rangle \notin \pi \wedge t(a) = t \\ \mathsf{mksbuff}(\mathsf{b}, t, \pi) & \text{otherwise} \end{cases}$$

$$\mathsf{chkatm}(n, \pi) \triangleq \begin{cases} \bot & \forall w. \begin{pmatrix} \mathtt{narR}\langle \_, \_, a, \_, w\rangle \in \pi \\ \wedge\ \overline{n}(a) = n \end{pmatrix} \implies \mathsf{B}\langle w\rangle \in \pi \\ \top & \text{otherwise} \end{cases}$$

$$\mathsf{mkpipe}(\mathbf{pipe}, t, \overline{n}, \varepsilon) \triangleq \mathbf{pipe}$$

$$\mathsf{mkpipe}(\mathbf{pipe}, t, \overline{n}, \pi{\cdot}\lambda) \triangleq \begin{cases} \mathsf{mkpipe}(a{\cdot}\mathbf{pipe}, t, \overline{n}, \pi) & \text{if} \begin{pmatrix} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathtt{NIC}\langle a\rangle \\ \wedge\ \mathtt{nlR}\langle \_, \_, a, \_\rangle \notin \pi \wedge\ \mathtt{nrR}\langle \_, \_, a, \_\rangle \notin \pi \\ \wedge\ \mathtt{nF}\langle a\rangle \notin \pi \wedge\ \mathtt{naF}\langle \_, \_, a, \_\rangle \notin \pi \\ \wedge\ \mathtt{narR}\langle \_, \_, a, \_, \_\rangle \notin \pi \end{pmatrix} \\ \mathsf{mkpipe}(w{\cdot}\mathbf{pipe}, t, \overline{n}, \pi) & \text{if} \begin{pmatrix} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathtt{NIC}\langle a\rangle \\ \wedge\ \mathtt{nlR}\langle \_, \_, a, w\rangle \in \pi \wedge\ \mathtt{nrW}\langle w, \_\rangle \notin \pi \end{pmatrix} \\ \mathsf{mkpipe}(e{\cdot}\mathbf{pipe}, t, \overline{n}, \pi) & \text{if} \begin{pmatrix} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathtt{NIC}\langle a\rangle \\ \wedge\ \mathtt{nlR}\langle \_, \_, a, w\rangle \in \pi \wedge\ \mathtt{nrW}\langle w, e\rangle \in \pi \\ \wedge\ \mathsf{CN}\langle e\rangle \notin \pi \end{pmatrix} \\ \mathsf{mkpipe}(w{\cdot}\mathbf{pipe}, t, \overline{n}, \pi) & \text{if} \begin{pmatrix} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathtt{NIC}\langle a\rangle \\ \wedge\ \mathtt{nrR}\langle \_, \_, a, w\rangle \in \pi \wedge\ \mathtt{nlW}\langle w, \_\rangle \notin \pi \end{pmatrix} \\ \mathsf{mkpipe}(w{\cdot}\mathbf{pipe}, t, \overline{n}, \pi) & \text{if} \begin{pmatrix} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathtt{NIC}\langle a\rangle \\ \wedge\ \mathtt{naF}\langle \_, \_, a, w\rangle \in \pi \wedge\ \mathtt{nlW}\langle w, \_\rangle \notin \pi \end{pmatrix} \\ \mathsf{mkpipe}(w{\cdot}w'{\cdot}\mathbf{pipe}, t, \overline{n}, \pi) & \text{if} \begin{pmatrix} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathtt{NIC}\langle a\rangle \\ \wedge\ \mathtt{narR}\langle \_, \_, a, w, w'\rangle \in \pi \wedge\ \mathtt{narW}\langle w'\rangle \notin \pi \end{pmatrix} \\ \mathsf{mkpipe}(\mathbf{pipe}, t, \overline{n}, \pi) & \text{otherwise} \end{cases}$$

$$\mathsf{mkwbR}(\mathbf{wb_R}, t, \overline{n}, \varepsilon) \triangleq \mathbf{wb_R}$$

$$\mathsf{mkwbR}(\mathbf{wb_R}, t, \overline{n}, \pi{\cdot}\lambda) \triangleq \begin{cases} \mathsf{mkwbR}(w{\cdot}\mathbf{wb_R}, t, \overline{n}, \pi) & \text{if} \begin{pmatrix} t(\lambda) = t \wedge \overline{n}(\lambda) = \overline{n} \wedge \mathsf{B}\langle w\rangle \notin \pi \\ \wedge\ \lambda \in \{\mathtt{nrW}\langle w, \_\rangle, \mathtt{narW}\langle w\rangle\} \end{pmatrix} \\ \mathsf{mkwbR}(\mathbf{wb_R}, t, \overline{n}, \pi) & \text{otherwise} \end{cases}$$

$$\mathsf{mkwbL}(\mathbf{wb_L}, t, \overline{n}, \varepsilon) \triangleq \mathbf{wb_L}$$

$$\mathsf{mkwbL}(\mathbf{wb_L}, t, \overline{n}, \pi{\cdot}\lambda) \triangleq \begin{cases} \mathsf{mkwbL}(e{\cdot}w{\cdot}\mathbf{wb_L}, t, \overline{n}, \pi) & \text{if} \begin{pmatrix} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathtt{nlW}\langle w, e\rangle \\ \wedge\ \mathsf{B}\langle w\rangle \notin \pi \wedge \mathtt{P}\langle \_, e\rangle \notin \pi \end{pmatrix} \\ \mathsf{mkwbL}(e{\cdot}\mathbf{wb_L}, t, \overline{n}, \pi) & \text{if} \begin{pmatrix} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathtt{nlW}\langle w, e\rangle \\ \wedge\ \mathsf{B}\langle w\rangle \in \pi \wedge \mathtt{P}\langle \_, e\rangle \notin \pi \end{pmatrix} \\ \mathsf{mkwbL}(e{\cdot}\mathbf{wb_L}, t, \overline{n}, \pi) & \text{if} \begin{pmatrix} t(\lambda) = t \wedge\ \overline{n}(\lambda) = \overline{n} \wedge\ \lambda = \mathsf{CN}\langle e\rangle \\ \wedge\ \mathtt{P}\langle \_, e\rangle \notin \pi \end{pmatrix} \\ \mathsf{mkwbL}(\mathbf{wb_L}, t, \overline{n}, \pi) & \text{otherwise} \end{cases}$$

Theorem 8. *For all* $\mathsf{P}, \mathsf{P}', \mathsf{M}, \mathsf{M}', \mathsf{B}, \mathsf{B}', \mathsf{A}, \mathsf{A}', \mathsf{QP}, \mathsf{QP}', \pi, \pi'$*:*

- $\mathsf{wf}(\mathsf{M}_0, \mathsf{B}_0, \mathsf{A}_0, \mathsf{QP}_0, \varepsilon)$*;*
- *if* $\mathsf{P}, \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi \Rightarrow \mathsf{P}', \mathsf{M}', \mathsf{B}', \mathsf{A}', \mathsf{QP}', \pi'$ *and* $\mathsf{wf}(\mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi)$ *then* $\mathsf{wf}(\mathsf{M}', \mathsf{B}', \mathsf{A}', \mathsf{QP}', \pi')$*;*
- *if* $\mathsf{P}, \mathsf{M}_0, \mathsf{B}_0, \mathsf{A}_0, \mathsf{QP}_0, \varepsilon \Rightarrow^* (\lambda t.\mathtt{skip}), \mathsf{M}, \mathsf{B}_0, \mathsf{A}_0, \mathsf{QP}, \pi$ *such that forall* $t, \overline{n}$ *we have* $\mathsf{QP}(t)(\overline{n}) = \langle \varepsilon, \varepsilon, \mathtt{nEX}^*\rangle$*, then* $\mathsf{wf}(\mathsf{M}, \mathsf{B}_0, \mathsf{A}_0, \mathsf{QP}, \pi)$ *and* $\mathsf{complete}(\pi)$*.*

The proof of the first part follows trivially from the definitions of M_0, B_0, A_0, and QP_0. The second part is proved by induction on the structure of $\Rightarrow$. The last part follows from the previous two parts and induction on the length of $\Rightarrow^*$, as well as how the definition of wf on empty store buffers and queue pairs (regardless of $\mathtt{nEX}$ in $\mathbf{wb_L}$) implies $\mathsf{complete}(\pi)$.

D.5 From Annotated Semantics to Declarative Semantics

We define

$$\mathsf{getEG}(\pi) \triangleq \begin{cases} (\mathsf{Event}, \mathsf{po}, \mathsf{rf}, \mathsf{pf}, \mathsf{mo}, \mathsf{nfo}, \mathsf{rao}) & \text{if } \mathsf{wfp}(\pi) \wedge \mathsf{complete}(\pi) \\ \text{undefined} & \text{otherwise} \end{cases}$$

with

$$\mathsf{Event} \triangleq \mathsf{Event}_0 \cup \{\mathsf{getA}(\lambda) \mid \lambda \in \pi\}$$

Recall that Event_0 is the set of initialisation events $\{init_x \mid x \in \mathsf{Loc}\}$, where $l(init_x) = \mathtt{lW}(x, 0)$

$$\mathsf{getA} : \mathsf{ALabel} \rightharpoonup \mathsf{Event}$$

$$\begin{aligned}
\mathsf{getA}(\mathtt{lR}\langle r, _\rangle) &\triangleq r \\
\mathsf{getA}(\mathtt{lW}\langle w \rangle) &\triangleq w \\
\mathsf{getA}(\mathtt{CAS}\langle u, _\rangle) &\triangleq u \\
\mathsf{getA}(\mathtt{F}\langle f \rangle) &\triangleq f \\
\mathsf{getA}(\mathtt{nlR}\langle r, _, _, _\rangle) &\triangleq r \\
\mathsf{getA}(\mathtt{nrW}\langle w \rangle) &\triangleq w \\
\mathsf{getA}(\mathtt{nrR}\langle r, _, _, _\rangle) &\triangleq r \\
\mathsf{getA}(\mathtt{nlW}\langle w, _\rangle) &\triangleq w \\
\mathsf{getA}(\mathtt{naF}\langle r, _, _, _\rangle) &\triangleq r
\end{aligned}$$

$$\begin{aligned}
\mathsf{getA}(\mathtt{narR}\langle r, _, _, _, _\rangle) &\triangleq r \\
\mathsf{getA}(\mathtt{narW}\langle w \rangle) &\triangleq w \\
\mathsf{getA}(\mathtt{P}\langle p, _\rangle) &\triangleq p \\
\mathsf{getA}(\mathtt{nF}\langle f \rangle) &\triangleq f \\
\mathsf{getA}(\mathsf{B}\langle w \rangle) &\triangleq w \\
\mathsf{getA}(\mathtt{Push}\langle _ \rangle) &\quad \text{is undefined} \\
\mathsf{getA}(\mathtt{NIC}\langle _ \rangle) &\quad \text{is undefined} \\
\mathsf{getA}(\mathsf{CN}\langle _ \rangle) &\quad \text{is undefined} \\
\mathsf{getA}(\mathcal{E}\langle _ \rangle) &\quad \text{is undefined}
\end{aligned}$$

We define $\mathsf{getI}\lambda(\_, \pi)$ and $\mathsf{getO}\lambda(\_, \pi)$ to perform the reverse operation of getA. In the case of write events, $\mathsf{getI}\lambda(\_, \pi)$ retrieves the first label sending the write to the buffer, while $\mathsf{getO}\lambda(\_, \pi)$ retrieves the second label committing the write to memory.

$$\mathsf{getI}\lambda(\_, \pi), \mathsf{getO}\lambda(\_, \pi) : \{\mathsf{getA}(\lambda) \mid \lambda \in \pi\} \to \mathsf{ALabel}$$

For all $\lambda \in \pi$:

- if $\mathsf{type}(\lambda) \in \{\mathtt{lR}, \mathtt{CAS}, \mathtt{F}, \mathtt{P}, \mathtt{nlR}, \mathtt{nrR}, \mathtt{narR}, \mathtt{naF}, \mathtt{nF}\}$,
  then $\mathsf{getI}\lambda(\mathsf{getA}(\lambda), \pi) \triangleq \mathsf{getO}\lambda(\mathsf{getA}(\lambda), \pi) \triangleq \lambda$;
- if $\mathsf{type}(\lambda) \in \{\mathtt{lW}, \mathtt{nlW}, \mathtt{nrW}, \mathtt{narW}\}$,
  then $\mathsf{getI}\lambda(\mathsf{getA}(\lambda), \pi) \triangleq \lambda$ while $\mathsf{getO}\lambda(\mathsf{getA}(\lambda), \pi) \triangleq \mathsf{B}\langle\lambda\rangle$.
- if $\lambda = \mathsf{B}\langle w \rangle$, then from $\mathsf{backComp}(\pi)$ there is $\lambda' \prec_\pi \lambda$ such that $\mathsf{type}(\lambda) \in \{\mathtt{lW}, \mathtt{nlW}, \mathtt{nrW}, \mathtt{narW}\}$ and $\mathsf{getA}(\lambda') = \mathsf{getA}(\lambda) = w$. From the previous case, we have $\mathsf{getI}\lambda(w, \pi) \triangleq \lambda'$ and $\mathsf{getO}\lambda(w, \pi) \triangleq \lambda$.

From this we define two relations IB and OB on Event total on all meaningful events by copying the ordering in π.

$$\mathsf{IB} \triangleq \{(e_1, e_2) \mid \mathsf{getIλ}(e_1, \pi) \prec_\pi \mathsf{getIλ}(e_2, \pi)\} \cup (\mathsf{Event}_0 \times (\mathsf{Event} \setminus \mathsf{Event}_0))$$

$$\mathsf{OB} \triangleq \{(e_1, e_2) \mid \mathsf{getOλ}(e_1, \pi) \prec_\pi \mathsf{getOλ}(e_2, \pi)\} \cup (\mathsf{Event}_0 \times (\mathsf{Event} \setminus \mathsf{Event}_0))$$

From $\mathsf{wfp}(\pi)$, IB and OB are transitive and irreflexive. Note: we could make IB and OB total by adding an arbitrary total order on Event_0.

$$\mathsf{rf} \triangleq \left\{ (w, r) \;\middle|\; \begin{array}{l} \mathtt{lR}\langle r, w\rangle \in \pi \vee \mathtt{nlR}\langle r, w, \_, \_\rangle \in \pi \vee \mathtt{nrR}\langle r, w, \_, \_\rangle \in \pi \vee \mathtt{CAS}\langle r, w\rangle \in \pi \\ \vee\ \mathtt{narR}\langle r, w, \_, \_, \_\rangle \in \pi \vee \mathtt{naF}\langle r, w, \_, \_\rangle \in \pi \end{array} \right\}$$

$$\mathsf{pf} \triangleq \left\{ (w, p) \;\middle|\; \begin{array}{l} \mathtt{nlW}\langle w, e\rangle \prec_\pi \mathtt{P}\langle p, e\rangle \\ \vee\ \mathtt{nrW}\langle w, e\rangle \prec_\pi \mathtt{P}\langle p, e\rangle \end{array} \right\}$$

$$\lambda \text{ generates } e \text{ in } \pi \triangleq \left(\begin{array}{l} \lambda \in \{\mathtt{lR}\langle e, \_\rangle, \mathtt{lW}\langle e\rangle, \mathtt{CAS}\langle e, \_\rangle, \mathtt{Push}\langle e\rangle, \mathtt{P}\langle e, \_\rangle, \mathtt{F}\langle e\rangle\} \\ \vee\ \lambda = \mathtt{Push}\langle a\rangle \wedge \left(\begin{array}{l} \lambda \prec_\pi \mathtt{nlR}\langle e, \_, a, \_\rangle \\ \vee\ \lambda \prec_\pi \mathtt{nlR}\langle \_, \_, a, e\rangle \\ \vee\ \lambda \prec_\pi \mathtt{nrR}\langle e, \_, a, \_\rangle \\ \vee\ \lambda \prec_\pi \mathtt{nrR}\langle \_, \_, a, e\rangle \\ \vee\ \lambda \prec_\pi \mathtt{naF}\langle e, \_, a, \_\rangle \\ \vee\ \lambda \prec_\pi \mathtt{naF}\langle \_, \_, a, e\rangle \\ \vee\ \lambda \prec_\pi \mathtt{narR}\langle e, \_, a, \_, \_\rangle \\ \vee\ \lambda \prec_\pi \mathtt{narR}\langle \_, \_, a, e, \_\rangle \\ \vee\ \lambda \prec_\pi \mathtt{narR}\langle \_, \_, a, \_, e\rangle \end{array} \right) \end{array} \right)$$

$$\mathsf{po} \triangleq \left(\begin{array}{l} \mathsf{Event}_0 \times (\mathsf{Event} \setminus \mathsf{Event}_0) \\ \cup \left\{ (e_1, e_2) \;\middle|\; \begin{array}{l} \lambda_1 \prec_\pi \lambda_2 \wedge t(\lambda_1) = t(\lambda_2) \\ \wedge\ \lambda_1 \text{ generates } e_1 \text{ in } \pi \\ \wedge\ \lambda_2 \text{ generates } e_2 \text{ in } \pi \end{array} \right\} \\ \cup \left\{ (r, w) \;\middle|\; \begin{array}{l} \mathtt{nlR}\langle r, \_, \_, w\rangle \in \pi \\ \vee\ \mathtt{nrR}\langle r, \_, \_, w\rangle \in \pi \\ \vee\ \mathtt{naF}\langle r, \_, \_, w\rangle \in \pi \\ \vee\ \mathtt{narR}\langle r, \_, \_, \_, w\rangle \in \pi \end{array} \right\} \\ \cup \{(w_1, w_2) \mid \mathtt{narR}\langle \_, \_, \_, w_2, w_1\rangle\} \in \pi \end{array} \right)$$

$$\mathsf{mo} \triangleq \left\{ (w_1, w_2) \;\middle|\; \begin{array}{l} w_1 = init_x \\ \wedge\ (\mathtt{B}\langle w_2\rangle \in \pi \vee \mathtt{CAS}\langle w_2, \_\rangle \in \pi) \\ \wedge\ \mathtt{loc}(w_1) = x = \mathtt{loc}(w_2) \end{array} \right\} \cup \left\{ (w_1, w_2) \;\middle|\; \begin{array}{l} \lambda_1 \prec_\pi \lambda_2 \\ \wedge\ \lambda_1 \in \{\mathtt{B}\langle w_1\rangle, \mathtt{CAS}\langle w_1, \_\rangle\} \\ \wedge\ \lambda_2 \in \{\mathtt{B}\langle w_2\rangle, \mathtt{CAS}\langle w_2, \_\rangle\} \\ \wedge\ \mathtt{loc}(w_1) = \mathtt{loc}(w_2) \end{array} \right\}$$

$$\mathsf{nfo} \triangleq \begin{pmatrix} \{(r,w) \mid \mathsf{sameqp}(r,w) \wedge \mathtt{nlR}\langle r,\_,\_,\_\rangle \prec_\pi \mathtt{nlW}\langle w,\_\rangle \prec_\pi \mathsf{B}\langle w\rangle\} \\ \cup\ \{(r,w) \mid \exists \lambda_r, \lambda_w.\mathsf{sameqp}(r,w) \wedge \lambda_r \prec_\pi \lambda_w \prec_\pi \mathsf{B}\langle w\rangle\} \\ \cup\ \{(w,r) \mid \mathsf{sameqp}(w,r) \wedge \mathtt{nlW}\langle w,\_\rangle \prec_\pi \mathsf{B}\langle w\rangle \prec_\pi \mathtt{nlR}\langle r,\_,\_,\_\rangle\} \\ \cup\ \{(w,r) \mid \exists \lambda_r, \lambda_w.\mathsf{sameqp}(w,r) \wedge \lambda_w \prec_\pi \mathsf{B}\langle w\rangle \prec_\pi \lambda_r\} \end{pmatrix}$$
$$\text{where } \lambda_r \in \{\mathtt{nrR}\langle r',\ldots\rangle, \mathtt{naF}\langle r',\ldots\rangle, \mathtt{narR}\langle r',\ldots\rangle\}$$
$$\lambda_w \in \{\mathtt{nrW}\langle w,\_\rangle, \mathtt{narW}\langle w\rangle\}$$

$$\mathsf{rao} \triangleq \left(\left\{ (r_1,r_2) \;\middle|\; \overline{n}(a_1) = \overline{n}(a_2) \wedge \begin{pmatrix} \lambda_1 \prec_\pi \lambda_2 \\ \wedge\ \lambda_1 \in \{\mathtt{naF}\langle r_1,a_1,\ldots\rangle, \mathtt{narR}\langle r_1,a_1,\ldots\rangle\} \\ \wedge\ \lambda_2 \in \{\mathtt{naF}\langle r_2,a_2,\ldots\rangle, \mathtt{narR}\langle r_2,a_2,\ldots\rangle\} \end{pmatrix} \right\}\right)$$

From an execution graph $E = \mathsf{getEG}(\pi)$, we use the definitions of the paper to define oppo, ippo, $\mathsf{rf_i}$, $\mathsf{rf_e}$, rb, $\mathsf{rb_i}$, ar, ob, and ib.

Lemma 2. $w \in \mathtt{nlW} \implies \exists r. \begin{pmatrix} r \in \mathtt{nrR} \wedge (r,w) \in \mathsf{po}|_{imm} \\ \vee\ r \in \mathtt{narR} \wedge (r,w) \in \mathsf{po}|_{imm} \cup (\mathsf{po}|_{imm})^2 \end{pmatrix}$

Proof. By definition of po, we can only have such $w \in \mathtt{nlW}$ if there is some $\lambda = \mathtt{Push}\langle a\rangle$ which generates w in π. Then we can consider the cases of a such that $\mathtt{Push}\langle a\rangle$ generates some $w \in \mathtt{nlW}$. Either:

- $a \in \mathtt{Put}$, then there is some $r \in \mathtt{nrR}$ with $(r,w) \in \mathsf{po}|_{\mathrm{imm}}$
- $a \in \mathtt{RCAS} \cup \mathtt{RFAA}$, then there is some $r \in \mathtt{narR}$ with either $(r,w) \in \mathsf{po}|_{\mathrm{imm}}$ (in the case of a failed RCAS) or $(r,w) \in (\mathsf{po}|_{\mathrm{imm}})^2$ (in the case of a successful RCAS or RFAA)

Theorem 9. $\mathsf{getEG}(\pi)$ *is well-formed.*

Proof. We need to check the conditions of a pre-execution (Def. 18) and of well-formedness (Def. 19). For the pre-execution conditions:

- Checking $\mathsf{Event}^0 \times (\mathsf{Event} \setminus \mathsf{Event}^0) \subseteq \mathsf{po}$:
  by definition.
- Checking po is a union of strict partial orders each on one thread:
  If $t(\mathsf{e}_1) \neq t(\mathsf{e}_2)$, then $(\mathsf{e}_1,\mathsf{e}_2) \notin \mathsf{po}$ and $(\mathsf{e}_2,\mathsf{e}_1) \notin \mathsf{po}$ by definition. If $t(\mathsf{e}_1) = t(\mathsf{e}_2)$, then either $(\mathsf{e}_1,\mathsf{e}_2) \in \mathsf{po}$ or $(\mathsf{e}_2,\mathsf{e}_1) \in \mathsf{po}$. This comes from the second case of the definition of po: if there is λ_1 and λ_2 such that λ_i generates e_i in π, then either $\lambda_1 \prec_\pi \lambda_2$ or $\lambda_2 \prec_\pi \lambda_1$.
- Checking that rf is functional on its range:
  If $r \in \mathcal{R} \subseteq \{\mathsf{getA}(\lambda) \mid \lambda \in \pi\}$, then we have either $\mathtt{lR}\langle r,\_\rangle$, $\mathtt{nlR}\langle r,\_,\_,\_\rangle$, $\mathtt{nrR}\langle r,\_,\_,\_\rangle$, $\mathtt{naF}\langle r,\_,\_,\_\rangle$, or $\mathtt{narR}\langle r,\_,\_,\_,\_\rangle$ in π, and r have at least one antecedent.
  If $(w,r) \in \mathsf{rf}$, let us assume $r \in \mathtt{nlR}$, then by definition $\mathtt{nlR}\langle r,w,\_,\_\rangle \in \pi$. Since $\mathsf{nodup}(\pi)$, for all $w' \neq w$, we have $\mathtt{nlR}\langle r,w',\_,\_\rangle \notin \pi$, and syntactically we cannot write $\mathtt{lR}\langle r,\_\rangle$ or $\mathtt{nrR}\langle r,\_,\_,\_\rangle$, so $(w',r) \notin \mathsf{rf}$. Similarly, $r \in \mathtt{lR}$, $r \in \mathtt{nrR}$, $r \in \mathtt{naF}$ or $r \in \mathtt{narR}$ only have one antecedent.

- Checking that rf relates events on the same location with matching values:
  By syntactic definition of the annotated labels $\mathtt{lR}$, $\mathtt{nlR}$, $\mathtt{nrR}$, $\mathtt{naF}$ and $\mathtt{narR}$, e.g., $\mathtt{lR}\langle r, w\rangle \implies \mathsf{eq}_{\mathtt{loc\&v}}(r, w)$.
- Checking that mo is a union of strict total orders for writes on each variables:
  By definition of mo, given that we have $\mathsf{complete}(\pi)$, e.g., if $\mathtt{lW}\langle w\rangle \in \pi$ then $\mathsf{B}\langle w\rangle \in \pi$.
- Checking that $\mathsf{pf} \subseteq \mathsf{po} \cap \mathtt{sqp}$:
  If $(w, p) \in \mathsf{pf}$ with $w \in \mathtt{nlW}$ (resp. $\mathtt{nrW}$), then we have $\mathtt{nlW}\langle w, e\rangle \prec_\pi \mathtt{P}\langle p, e\rangle$. There is λ such that λ generates w in π, and we have $\lambda \prec_\pi \mathtt{nlW}\langle w, e\rangle \prec_\pi \mathtt{P}\langle p, e\rangle$. Also, $t(p) = t(w)$ and $\overline{n}(p) = \overline{n}(e) = \overline{n}(w)$, so we have $(w, p) \in \mathsf{po}$ and $(w, p) \in \mathtt{sqp}$.
- Checking that pf is functional on its domain:
  If $(w, p) \in \mathsf{pf}$ with $w \in \mathtt{nlW}$ (resp. $\mathtt{nrW}$), then we have $\mathtt{nlW}\langle w, e\rangle \prec_\pi \mathtt{P}\langle p, e\rangle$. From $\mathsf{nodup}(\pi)$, for all $p' \neq p$ we have $\mathtt{P}\langle p, e\rangle \notin \pi$, so w has at most one image.
- Checking that pf is total and functional on its range:
  If $p \in \mathsf{Event}$, then there is $e \in \mathtt{nlEX}$ (resp. $\mathtt{nrEX}$) such that $\mathtt{P}\langle p, e\rangle \in \pi$. From $\mathsf{backComp}(\pi)$ there is $w \in \mathtt{nlW}$ (resp. $\mathtt{nrW}$) such that $\mathtt{nlW}\langle w, e\rangle \prec_\pi \mathtt{P}\langle p, e\rangle$, and so $(w, p) \in \mathsf{pf}$. From $\mathsf{nodup}(\pi)$, e cannot be used in another $\mathtt{nlW}$ (resp. $\mathtt{nrW}$) annotated label, and p has exactly one antecedent.
- Checking that for all $(a, b) \in \mathtt{sqp}$, $a \in \mathtt{nrR} \cup \mathtt{naF} \cup \mathtt{narR}$, $b \in \mathtt{nrW} \cup \mathtt{narW}$, (resp. $\mathtt{nlR}/\mathtt{nlW}$) then $(a, b) \in \mathsf{nfo} \cup \mathsf{nfo}^{-1}$:
  By definition of nfo, given that $\mathsf{bufFlushOrd}(\pi)$ forbids such interleavings as $\mathtt{nrW}\langle w, \_\rangle \prec_\pi \mathtt{nrR}\langle r, \_, \_, \_\rangle \prec_\pi \mathsf{B}\langle w\rangle$ (resp. $\mathtt{nlW}$ and $\mathtt{nlR}$) when $\mathsf{sameqp}(r, w)$.
- Checking that rao is a union of strict total orders for remote atomic reads:
  By definition of rao.

For the well-formedness conditions:

(1) Let us assume $(w_1, w_2) \in \mathsf{po} \cap \mathtt{sqp}$ and $(w_2, p_2) \in \mathsf{pf}$. The three events are on the same thread and queue pair.
If $w_1 \in \mathtt{nlW}$, then by $\mathsf{complete}(\pi)$ there is a chain $\mathtt{Push}\langle a_1\rangle \prec_\pi \mathtt{NIC}\langle a_1\rangle \prec_\pi \mathtt{nR} \prec_\pi \mathtt{nlW}\langle w_1, e_1\rangle$ for some $\mathtt{nR} \in \{\mathtt{nrR}\langle \_, \_, a_1, w_1\rangle, \mathtt{naF}\langle \_, \_, a_1, w_1\rangle, \mathtt{narR}\langle \_, \_, a_1, w_1, \_\rangle\}$; if $w_1 \in \mathtt{nrW}$, there is instead a chain $\mathtt{Push}\langle a_1\rangle \prec_\pi \mathtt{NIC}\langle a_1\rangle \prec_\pi \mathtt{nlR}\langle \_, \_, a_1, w_1\rangle \prec_\pi \mathtt{nrW}\langle w_1, e_1\rangle \prec_\pi \mathsf{CN}\langle e_1\rangle$. Similarly there is a chain for w_2. By $(w_1, w_2) \in \mathsf{po}$ we have $\mathtt{Push}\langle a_1\rangle \prec_\pi \mathtt{Push}\langle a_2\rangle$, and by $\mathsf{bufFlushOrd}(\pi)$ we have $\mathtt{NIC}\langle a_1\rangle \prec_\pi \mathtt{NIC}\langle a_2\rangle$.
Let us call λ_1 the last annotated label on the chain for w_1, i.e., either $\mathtt{nlW}\langle w_1, e_1\rangle$ or $\mathsf{CN}\langle e_1\rangle$. Similarly, λ_2 is the last annotated label on the chain for w_2. There are four cases to consider, but in all four $\mathsf{nicActOrder}(\pi)$ implies $\lambda_1 \prec_\pi \lambda_2$.
Then, from $\mathsf{pollOrder}(\pi)$, there is p_1 such that $\mathtt{P}\langle p_1, e_1\rangle \prec_\pi \mathtt{P}\langle p_2, e_2\rangle$. By definitions, we have both $(w_1, p_1) \in \mathsf{pf}$ and $(p_1, p_2) \in \mathsf{po}$.

(2) If $r \in \mathtt{nlR}$, then there is $w \in \mathtt{nrW}$ (taken from $\mathtt{nlR}\langle r, \_, \_, w\rangle$) such that $(r, w) \in \mathsf{po}|_{\mathrm{imm}}$. This is by the last case of definition of po, since there is λ_a such that we have both λ_a generates r in π and λ_a generates w in π.
Similarly for $\mathtt{nrR}/\mathtt{nlW}$ and $\mathtt{nrW}/\mathtt{nlR}$.

(3) If $(r, w) \in \mathsf{po}|_{\mathrm{imm}}$, $\mathsf{type}(r) \in \{\mathtt{nlR}, \mathtt{nrR}\}$, and $\mathsf{type}(w) \in \{\mathtt{nlW}, \mathtt{nrW}\}$, then $(r, w) \in \mathsf{po}$ comes from the third case of the definition of po, and we have either $\mathtt{nlR}\langle r, \_, \_, w\rangle$ or $\mathtt{nrR}\langle r, \_, \_, w\rangle$ in π. In both cases, we have $v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w)$ by syntactic definition of the annotated labels.

(4) (a) If $r \in \mathtt{narR}$, then either: There is $\mathtt{naF}\langle r, \_, \_, w\rangle \in \pi$ such that $w \in \mathtt{nlW}$ and $(r, w) \in \mathsf{po}|_{\mathrm{imm}}$. This follows from the second case definition of po. There is $\mathtt{narR}\langle r, \_, \_, w_2, w_1\rangle \in \pi$ such that $w_1 \in \mathtt{narW}$, $w_2 \in \mathtt{nlW}$, and $(r, w_1), (w_1, w_2) \in \mathsf{po}|_{\mathrm{imm}}$. This follows from the second and third cases of the definition of po since there is λ_a which generates r, w_1 and w_2 in π. (b) If $w \in \mathtt{narW}$ then $(r, w), (w, w') \in \mathsf{po}|_{\mathrm{imm}}$ with $r \in \mathtt{narR}$ and $w' \in \mathtt{nlW}$ comes from the second case definition of po.

(5) If $(r, w) \in G.\mathsf{po}|_{\mathrm{imm}}$, $\mathsf{type}(r) = \mathtt{narR}$ and $\mathsf{type}(w) = \mathtt{nlW}$, then (r, w) comes from the second case definition of po and we have $\mathtt{naF}\langle r, \_, \_, w\rangle \in \pi$. Then $v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w)$ by the syntax of annotated labels. If $(r, w_1), (w_1, w_2) \in G.po|_{\mathrm{imm}}$, $\mathsf{type}(r) = \mathtt{narR}$, $\mathsf{type}(w_1) = \mathtt{narW}$ and $\mathsf{type}(w_2) = \mathtt{nlW}$, then (r, w_1) comes from the second case definition of po and (w_1, w_2) from the third case, so we have $\mathtt{narR}\langle r, \_, \_, w_2, w_1\rangle \in \pi$. Then $v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w_2)$ by the syntax of annotated labels.

(6) Comes from Lem. 2.

Lemma 3. $\mathsf{OB}; [\mathtt{Inst}] \subseteq \mathsf{IB}$ *and* $[\mathtt{Inst}]; \mathsf{IB} \subseteq \mathsf{OB}$.

Proof. If $(e_1, e_2) \in \mathsf{OB}; [\mathtt{Inst}]$, then $\mathsf{getO}\lambda(e_1, \pi) \prec_\pi \mathsf{getO}\lambda(e_2, \pi) = \mathsf{getI}\lambda(e_2, \pi)$.

- If $e_1 \in \mathtt{Inst}$, then $\mathsf{getO}\lambda(e_1, \pi) = \mathsf{getI}\lambda(e_1, \pi)$, so we have $\mathsf{getI}\lambda(e_1, \pi) \prec_\pi \mathsf{getI}\lambda(e_2, \pi)$ and $(e_1, e_2) \in \mathsf{IB}$.
- If $e_1 \in \{\mathtt{lW}, \mathtt{nlW}, \mathtt{nrW}, \mathtt{narW}\}$, there is λ such that $\mathsf{type}(\lambda) \in \{\mathtt{lW}, \mathtt{nlW}, \mathtt{nrW}, \mathtt{narW}\}$, $\mathsf{getA}(\lambda) = e_1$, and $\mathsf{getI}\lambda(e_1, \pi) = \lambda \prec_\pi \mathsf{B}\langle e_1\rangle = \mathsf{getO}\lambda(e_1, \pi)$. By transitivity we again have $\mathsf{getI}\lambda(e_1, \pi) \prec_\pi \mathsf{getI}\lambda(e_2, \pi)$ and $(e_1, e_2) \in \mathsf{IB}$.

With a similar reasoning, we can see that $[\mathtt{Inst}]; \mathsf{IB} \subseteq \mathsf{OB}$.

Theorem 10. $\mathsf{getEG}(\pi)$ *is consistent.*

Proof. From Definition 20, we need to check that both ib and ob are irreflexive. Since IB and OB are irreflexive, it is enough to show that $\mathsf{ib} \subseteq \mathsf{IB}$ and $\mathsf{ob} \subseteq \mathsf{OB}$.

The explicit definition using limits is the following (where $\mathsf{rf_e} \triangleq (\mathsf{rf} \setminus \mathsf{rf_i})$ includes $(\mathsf{rf} \cap \mathtt{sqp})$ since we assume the PCIe guarantees hold):

$$\mathsf{ib}^0 \triangleq (\mathsf{ippo} \cup \mathsf{rf} \cup \mathsf{pf} \cup \mathsf{rb_i} \cup \mathsf{nfo})^+$$
$$\mathsf{ob}^0 \triangleq (\mathsf{oppo} \cup \mathsf{rf_e} \cup [\mathtt{nlW}]; \mathsf{pf} \cup \mathsf{rb} \cup \mathsf{nfo} \cup \mathsf{mo} \cup \mathsf{rao} \cup \mathsf{ar}; \mathsf{rao})^+$$
$$\mathsf{ib}^{n+1} \triangleq (\mathsf{ib}^n \cup \mathsf{ob}^n; [\mathtt{Inst}])^+$$
$$\mathsf{ob}^{n+1} \triangleq (\mathsf{ob}^n \cup [\mathtt{Inst}]; \mathsf{ib}^n)^+$$
$$\mathsf{ib} \triangleq \lim_{n\to\infty} \mathsf{ib}^n$$
$$\mathsf{ob} \triangleq \lim_{n\to\infty} \mathsf{ob}^n$$

It is then enough to show that $\mathsf{ib}^0 \subseteq \mathsf{IB}$ and $\mathsf{ob}^0 \subseteq \mathsf{OB}$. Using Lemma 3 above, we can check the induction case:

$$\mathsf{ib}^{n+1} = (\mathsf{ib}^n \cup \mathsf{ob}^n; [\mathtt{Inst}])^+ \subseteq (\mathsf{ib}^n \cup \mathsf{OB}; [\mathtt{Inst}])^+ \subseteq (\mathsf{IB} \cup \mathsf{IB})^+ = \mathsf{IB}$$
$$\mathsf{ob}^{n+1} = (\mathsf{ob}^n \cup [\mathtt{Inst}]; \mathsf{ib}^n)^+ \subseteq (\mathsf{ob}^n \cup [\mathtt{Inst}]; \mathsf{IB})^+ \subseteq (\mathsf{OB} \cup \mathsf{OB})^+ = \mathsf{OB}$$

Since IB and OB are transitive, we need to check the components of ib^0 and ob^0. There are twelve cases to verify.

- Checking $\mathsf{ippo} \subseteq \mathsf{IB}$.
  Let $E^{\mathrm{cpu}} = \{\mathtt{lR}, \mathtt{lW}, \mathtt{CAS}, \mathtt{F}, \mathtt{P}\}$ and $E^{\mathrm{nic}} = \{\mathtt{nlR}, \mathtt{nrR}, \mathtt{narR}, \mathtt{naF}, \mathtt{nlW}, \mathtt{nrW}, \mathtt{narW}, \mathtt{nF}\}$.
  $[E^{\mathrm{cpu}}]; \mathsf{po} \subseteq \mathsf{IB}$ by definition of po and IB: E^{cpu} are the events for which the same annotated label is used to define po and IB, i.e., $\forall e \in E^{\mathrm{cpu}}$, $\mathsf{getl}\lambda(e, \pi)$ generates e in π.
  To check that $[E^{\mathrm{nic}}]; \mathsf{ippo}; [E^{\mathrm{nic}}] \subseteq \mathsf{IB}$, there are 36 cases to consider. They are all trivially satisfied by $\mathsf{nicActOrder}(\pi)$ and $\mathsf{backComp}(\pi)$.
- Checking $\mathsf{oppo} \subseteq \mathsf{OB}$.
  From above we have $[\mathtt{Inst}]; \mathsf{oppo} \subseteq [\mathtt{Inst}]; \mathsf{ippo} \subseteq [\mathtt{Inst}]; \mathsf{IB} \subseteq \mathsf{OB}$.
  $[\mathtt{lW}]; \mathsf{po}; [\mathsf{Event} \setminus (\mathtt{lR} \cup \mathtt{P})] \subseteq \mathsf{OB}$ by using $\mathsf{bufFlushOrd}(\pi)$.
  For the remaining cases:

(G7) $[\mathtt{nrW}]; (\mathsf{po} \cap \mathtt{sqp}); [\mathtt{nrW}] \subseteq \mathsf{OB}$ comes from $\mathsf{nicActOrder}(\pi)$ (i.e., $\mathtt{nrW}\langle\ldots\rangle \prec_\pi \mathtt{nrW}\langle\ldots\rangle$) and $\mathsf{bufFlushOrd}(\pi)$ (i.e., $\mathsf{B}\langle\ldots\rangle \prec_\pi \mathsf{B}\langle\ldots\rangle$).

(G8) $[\mathtt{nrW}]; (\mathsf{po} \cap \mathtt{sqp}); [\mathtt{narR}] \subseteq \mathsf{OB}$ comes from $\mathsf{nicActOrder}(\pi)$ (i.e., $\mathtt{nrW}\langle\ldots\rangle \prec_\pi \mathtt{narR}\langle\ldots\rangle$) and $\mathsf{bufFlushOrd}(\pi)$ (i.e., $\mathsf{B}\langle\ldots\rangle \prec_\pi \mathtt{narR}\langle\ldots\rangle$).

(G9) If $e_1 \in \mathtt{nrW}$, $e_3 \in \mathtt{narW}$, and $(e_1, e_3) \in (\mathsf{po} \cap \mathtt{sqp})$, then from Def. 19 there is $e_2 \in \mathtt{narR}$ such that $(e_2, e_3) \in \mathsf{po}|_{\mathrm{imm}}$ and thus $(e_1, e_2) \in (\mathsf{po} \cap \mathtt{sqp})$. From case G8 above, we have $(e_1, e_2) \in \mathsf{OB}$. From $\mathsf{backComp}(\pi)$, we have $(e_2, e_3) \in [\mathtt{Inst}]; \mathsf{IB} \subseteq \mathsf{OB}$. Thus $[\mathtt{nrW}]; (\mathsf{po} \cap \mathtt{sqp}); [\mathtt{narW}] \subseteq \mathsf{OB}$.

(G10) $[\mathtt{nrW}]; (\mathsf{po} \cap \mathtt{sqp}); [\mathtt{nrR}] \subseteq \mathsf{OB}$ comes from $\mathsf{nicActOrder}(\pi)$ (i.e., $\mathtt{nrW}\langle\ldots\rangle \prec_\pi \mathtt{nrR}\langle\ldots\rangle$) and $\mathsf{bufFlushOrd}(\pi)$ (i.e., $\mathtt{nrW}\langle\ldots\rangle \prec_\pi \mathsf{B}\langle\ldots\rangle \prec_\pi \mathtt{nrR}\langle\ldots\rangle$).

(G11) If $e_1 \in \mathtt{nrW}$, $e_3 \in \mathtt{nlW}$, and $(e_1, e_3) \in (\mathsf{po} \cap \mathtt{sqp})$, then from Def. 19 there is $e_2 \in (\mathtt{narR} \cup \mathtt{nrR})$ such that $(e_2, e_3) \in \mathsf{po}|_{\mathrm{imm}}^{\{1,2\}}$ and thus $(e_1, e_2) \in (\mathsf{po} \cap \mathtt{sqp})$. Then $(e_1, e_2) \subseteq \mathsf{OB}$ comes from cases G9 and G10 respectively. From $\mathsf{backComp}(\pi)$, we have $(e_2, e_3) \in [\mathtt{Inst}]; \mathsf{IB} \subseteq \mathsf{OB}$. Thus $[\mathtt{nrW}]; (\mathsf{po} \cap \mathtt{sqp}); [\mathtt{nlW}] \subseteq \mathsf{OB}$.

(I7) $[\mathtt{narW}]; (\mathsf{po} \cap \mathtt{sqp}); [\mathtt{nrW}] \subseteq \mathsf{OB}$ comes from $\mathsf{nicActOrder}(\pi)$ (i.e., $\mathtt{narW}\langle\ldots\rangle \prec_\pi \mathtt{nrW}\langle\ldots\rangle$) and $\mathsf{bufFlushOrd}(\pi)$ (i.e., $\mathsf{B}\langle\ldots\rangle \prec_\pi \mathsf{B}\langle\ldots\rangle$).

(I8) $[\mathtt{narW}]; (\mathsf{po} \cap \mathtt{sqp}); [\mathtt{narR}] \subseteq \mathsf{OB}$ follows from Def. 19, $\mathsf{nicActOrder}(\pi)$ and $\mathsf{bufFlushOrd}(\pi)$ by similar reasoning to I7.

(I9) $[\mathtt{narW}]; (\mathsf{po} \cap \mathtt{sqp}); [\mathtt{narW}] \subseteq \mathsf{OB}$ follows similarly to I7.

(I10) $[\mathtt{narW}]; (\mathsf{po} \cap \mathtt{sqp}); [\mathtt{nrR}] \subseteq \mathsf{OB}$ follows similarly to I7.

(K11) $[\mathtt{nlW}]; (\mathsf{po} \cap \mathtt{sqp}); [\mathtt{nlW}] \subseteq \mathsf{OB}$ comes from $\mathsf{nicActOrder}(\pi)$ (i.e., $\mathtt{nlW}\langle\ldots\rangle \prec_\pi \mathtt{nlW}\langle\ldots\rangle$) and $\mathsf{bufFlushOrd}(\pi)$ (i.e., $\mathsf{B}\langle\ldots\rangle \prec_\pi \mathsf{B}\langle\ldots\rangle$).

- Checking $\mathsf{rf_e} \subseteq \mathsf{OB}$.
  If $(w, r) \in \mathsf{rf_e}$, there is π_1 and π_2 such that $\pi = \pi_2 \cdot \mathsf{getO}\lambda(r, \pi) \cdot \pi_1$, and we use $\mathsf{wfrd}(\pi)$.

- If $r \in \texttt{lR}$, we have $\mathsf{wfrdCPU}(r, w, \pi_1)$. The definition allow for three different cases. In the first case, $\lambda \in \{\mathsf{B}\langle w\rangle, \texttt{CAS}\langle w, \_\rangle\}$ is in π_1; we have $\lambda = \mathsf{getO}\lambda(w, \pi) \prec_\pi \mathsf{getO}\lambda(r, \pi)$ and so $(w, r) \in \mathsf{OB}$. In the second case, we have $\lambda = \texttt{lW}\langle w\rangle$ and $t(w) = t(r)$; so $(w, r) \in [\texttt{lW}]; (\mathsf{rf} \cap \mathsf{sthd}); [\texttt{lR}] = \mathsf{rf_i}$, which contradicts $(w, r) \in \mathsf{rf_e} = \mathsf{rf} \setminus \mathsf{rf_i}$. In the third case, $w = init_x$ for some location x, so $(w, r) \in \mathsf{Event}_0 \times (\mathsf{Event} \setminus \mathsf{Event}_0) \subseteq \mathsf{OB}$.
- If $r \in \texttt{CAS}$, similarly to above, except the second case of $\mathsf{wfrdCPU}(r, w, \pi_1)$ is not possible because of $\mathsf{bufFlushOrd}(\pi)$: $\mathsf{B}\langle w\rangle \notin \pi_1$ while CAS acts as a memory fence.
- If $r \in \texttt{nlR}$, we have $\mathsf{wfrdNIC}(r, w, \pi_1)$, with two possibilities. In the first case, $\lambda \in \{\mathsf{B}\langle w\rangle, \texttt{CAS}\langle w, \_\rangle\}$ is in π_1; we have $\lambda = \mathsf{getO}\lambda(w, \pi) \prec_\pi \mathsf{getO}\lambda(r, \pi)$ and so $(w, r) \in \mathsf{OB}$. In the second case, $w = init_x$ for some location x, so $(w, r) \in \mathsf{Event}_0 \times (\mathsf{Event} \setminus \mathsf{Event}_0) \subseteq \mathsf{OB}$.
- If $r \in \texttt{nrR}$ or $\texttt{narR}$, similarly to above.

- Checking $\mathsf{rf} \subseteq \mathsf{IB}$.
  From above we have $\mathsf{rf_e} = \mathsf{rf_e}; [\texttt{Inst}] \subseteq \mathsf{OB}; [\texttt{Inst}] \subseteq \mathsf{IB}$.
  If $(w, r) \in \mathsf{rf_i} \subseteq [\texttt{lW}]; \mathsf{rf}; [\texttt{lR}]$, then there is $\texttt{lR}\langle r, w\rangle \in \pi$. There is π_1 and π_2 such that $\pi = \pi_2 \cdot \texttt{lR}\langle r, w\rangle \cdot \pi_1$. So by $\mathsf{wfrd}(\pi)$ we have $\mathsf{wfrdCPU}(r, w, \pi_1)$ which implies $\texttt{lW}\langle w\rangle \prec_\pi \texttt{lR}\langle r, w\rangle$ and $(w, r) \in \mathsf{IB}$.
- Checking $[\texttt{nlW}]; \mathsf{pf} \subseteq \mathsf{OB}$.
  If $(w, p) \in \mathsf{pf}$ with $w \in \texttt{nlW}$, then there exists e such that $\texttt{nlW}\langle w, e\rangle \prec_\pi \texttt{P}\langle p, e\rangle$. From $\mathsf{backComp}(\pi)$, we have $\texttt{nlW}\langle w, e\rangle \prec_\pi \mathsf{B}\langle w\rangle \prec_\pi \texttt{P}\langle p, e\rangle$ and so $(w, p) \in \mathsf{OB}$.
- Checking $\mathsf{pf} \subseteq \mathsf{IB}$.
  If $(w, p) \in \mathsf{pf}$, then there exists e such that either $\texttt{nlW}\langle w, e\rangle \prec_\pi \texttt{P}\langle p, e\rangle$ or $\texttt{nrW}\langle w, e\rangle \prec_\pi \texttt{P}\langle p, e\rangle$. In both cases we immediately have $(w, p) \in \mathsf{IB}$.
- Checking $\mathsf{rb_i} \subseteq \mathsf{IB}$.
  If $(r, w') \in \mathsf{rb_i}$ then $r \in \texttt{lR}$, $w' \in \texttt{lW}$, $t(r) = t(w')$, and there exists w such that $(w, r) \in \mathsf{rf}$ and $(w, w') \in \mathsf{mo}$. There is π_4 and π_3 such that $\pi = \pi_4 \cdot \texttt{lR}\langle r, w\rangle \cdot \pi_3$. So by $\mathsf{wfrd}(\pi)$ we have $\mathsf{wfrdCPU}(r, w, \pi_3)$, and there is three cases to consider.
  - In the first case, $\pi_3 = \pi_2 \cdot \lambda_w \cdot \pi_1$, with $\lambda_w \in \{\mathsf{B}\langle w\rangle, \texttt{CAS}\langle w, \_\rangle\}$, and $\mathsf{B}\langle w'\rangle \notin \pi_2$. Since $(w, w') \in \mathsf{mo}$ we have $\mathsf{B}\langle w'\rangle \notin \pi_1$, an so $\mathsf{B}\langle w'\rangle \notin \pi_3$. The last condition of the first case then gives us $\texttt{lW}\langle w'\rangle \notin \pi_3$, which implies $(r, w') \in \mathsf{IB}$.
  - In the second case, $\pi_3 = \pi_2 \cdot \lambda_w \cdot \pi_1$, with $\lambda_w = \texttt{lW}\langle w\rangle$, $\mathsf{thread}(w) = \mathsf{thread}(r)$, and $\mathsf{B}\langle w\rangle \notin \pi_3$. Then w and w' are on the same thread, and by $\mathsf{bufFlushOrd}(\pi)$ and $(w, w') \in \mathsf{mo}$ we have $\texttt{lW}\langle w\rangle \prec_\pi \texttt{lW}\langle w'\rangle$ and $\texttt{lW}\langle w'\rangle \notin \pi_1$. The last condition of the second case gives us $\texttt{lW}\langle w'\rangle \notin \pi_2$, so $\texttt{lW}\langle w'\rangle \notin \pi_3$ and $(r, w') \in \mathsf{IB}$.
  - In the last case, $w = init_x$ for some location x, and we immediately get $\texttt{lW}\langle w'\rangle \notin \pi_3$, which implies $(r, w') \in \mathsf{IB}$.
- Checking $\mathsf{rb} \subseteq \mathsf{OB}$.
  If $(r, w') \in \mathsf{rb}$, then there exists w such that $(w, r) \in \mathsf{rf}$ and $(w, w') \in \mathsf{mo}$. By definition of rf, there is π_4 and π_3 such that $\pi = \pi_4 \cdot \lambda_r \cdot \pi_3$, with $\lambda_r \in \{\texttt{lR}\langle r, w\rangle, \texttt{CAS}\langle r, w\rangle, \texttt{nlR}\langle r, w, \_, \_\rangle, \texttt{nrR}\langle r, w, \_, \_\rangle, \texttt{naF}\langle r, w, \_, \_\rangle, \texttt{narR}\langle r, w, \_, \_, \_\rangle\}$. So by $\mathsf{wfrd}(\pi)$ we have either $\mathsf{wfrdNIC}(r, w, \pi_3)$ or $\mathsf{wfrdCPU}(r, w, \pi_3)$, and there are five cases to consider.

  - In the first case of $\mathsf{wfrdNIC}(r, w, \pi_3)$, $\pi_3 = \pi_2 \cdot \mathsf{getO}\lambda(w, \pi) \cdot \pi_1$, and $\mathsf{getO}\lambda(w', \pi) \notin \pi_2$. Since $(w, w') \in \mathsf{mo}$ we have $\mathsf{getO}\lambda(w', \pi) \notin \pi_1$, and thus $\mathsf{getO}\lambda(w', \pi) \notin \pi_3$. So $\mathsf{getO}\lambda(w', \pi) \in \pi_4$ and $(r, w') \in \mathsf{OB}$.
  - In the last case $\mathsf{wfrdNIC}(r, w, \pi_3)$, $w = init_x$ for some location x, and we immediately have $\mathsf{getO}\lambda(w', \pi) \notin \pi_3$, which implies $(r, w') \in \mathsf{OB}$.
  - For the first case of $\mathsf{wfrdCPU}(r, w, \pi_3)$, same reasoning as for the first case of wfrdNIC.
  - For the second case of $\mathsf{wfrdCPU}(r, w, \pi_3)$, $\pi_3 = \pi_2 \cdot \mathsf{getI}\lambda(w, \pi) \cdot \pi_1$, with $\mathsf{thread}(w) = \mathsf{thread}(r)$, and $\mathsf{getO}\lambda(w, \pi) \notin \pi_3$. So $\mathsf{getO}\lambda(w, \pi) \in \pi_4$, and since $(w, w') \in \mathsf{mo}$ we have $\mathsf{getO}\lambda(w', \pi) \in \pi_4$ as well, and $(r, w') \in \mathsf{OB}$.
  - For the last case of $\mathsf{wfrdCPU}(r, w, \pi_3)$, same reasoning as for the last case of wfrdNIC.
- Checking $\mathsf{nfo} \subseteq \mathsf{IB}$.
  By definition of nfo.
- Checking $\mathsf{nfo} \subseteq \mathsf{OB}$.
  By definition of nfo.
- Checking $\mathsf{mo} \subseteq \mathsf{OB}$.
  By definition of mo, as what matters are the $init_x$, $\mathsf{B}\langle w \rangle$, and $\mathtt{CAS}\langle w, \_ \rangle$ events.
- Checking $\mathsf{rao} \subseteq \mathsf{OB}$.
  By definition of rao.
- Checking $\mathsf{ar}; \mathsf{rao} \subseteq \mathsf{OB}$.
  If $(w, r_1) \in \mathsf{ar}$ then $\mathtt{narR}\langle r_1, a_1, \_, \_, w \rangle \in \pi$ for some $a_1 \in \mathtt{rRMW}$, and if $(r_1, r_2) \in \mathsf{rao}$ then $\mathtt{narR}\langle r_1, \_, a_1, \_, w \rangle \prec_\pi \lambda_r$ for some $\lambda_r \in \{\mathtt{naF}\langle r_2, \_, a_2, \_ \rangle, \mathtt{narR}\langle r_2, \_, a_2, \_, \_ \rangle\}$, with $\overline{n}(a_1) = \overline{n}(a_2)$. Then using $\mathsf{nicAtomicity}(\pi)$ we have that $\mathsf{B}\langle w \rangle \prec_\pi \lambda_r$.

D.6 From Declarative Semantics to Annotated Semantics

From a program P and a well-formed consistent execution graph $G = (\mathsf{Event}, \mathsf{po}, \mathsf{rf}, \mathsf{pf}, \mathsf{mo}, \mathsf{nfo}, \mathsf{rao})$, where $(\mathsf{Event}, \mathsf{po})$ is generated by P, we want to reconstruct an annotated semantics execution.

Theorem 11. ib *and* ob *can be extended into total relations* IB *and* OB *on* Event *such that:*

- IB *and* OB *are irreflexive and transitive;*
- $\mathsf{OB}; [\mathtt{Inst}] \subseteq \mathsf{IB}$ *and* $[\mathtt{Inst}]; \mathsf{IB} \subseteq \mathsf{OB}$.

Proof. We show that if ib is not already total we can extend it (and maybe ob) into a strictly bigger relation satisfying the constraints of the theorem. Let us assume that there is $(a, b) \in \mathsf{Event}^2$ such that $(a, b) \notin \mathsf{ib}$ and $(b, a) \notin \mathsf{ib}$. We arbitrarily decide to include (a, b) in our relation and we define $\mathsf{ib}' = (\mathsf{ib} \cup \{(a, b)\})^+$ and $\mathsf{ob}' = (\mathsf{ob} \cup [\mathtt{Inst}]; \mathsf{ib}')^+$.

Clearly ib' and ob' are transitive, ib' is irreflexive, and $[\mathtt{Inst}]; \mathsf{ib}' \subseteq \mathsf{ob}'$. We need to prove the following two facts: ob' is still irreflexive; and $\mathsf{ob}'; [\mathtt{Inst}] \subseteq \mathsf{ib}'$.

First, let us check that $(\mathsf{ob} \cup [\mathtt{Inst}]; \mathsf{ib}')^+$ is irreflexive. Since ob and $([\mathtt{Inst}]; \mathsf{ib}')$ are both transitive and irreflexive, a cycle would only be possible by alternating between the two components, so it is enough to show that $(\mathsf{ob}; ([\mathtt{Inst}]; \mathsf{ib}'))^+$ is irreflexive. But $(\mathsf{ob}; ([\mathtt{Inst}]; \mathsf{ib}'))^+ = ((\mathsf{ob}; [\mathtt{Inst}]); \mathsf{ib}')^+ \subseteq (\mathsf{ib}; \mathsf{ib}')^+ \subseteq \mathsf{ib}'$ is irreflexive. Thus ob' is irreflexive.

Then, we need to check that $\mathsf{ob}'; [\mathtt{Inst}] \subseteq \mathsf{ib}'$. Using some rewriting, $\mathsf{ob}' = (\mathsf{ob} \cup [\mathtt{Inst}]; \mathsf{ib}')^+ = \mathsf{ob} \cup (\mathsf{ob}^*; ([\mathtt{Inst}]; \mathsf{ib}'))^+; \mathsf{ob}^*$. We know $\mathsf{ob}; [\mathtt{Inst}] \subseteq \mathsf{ib}'$, which also implies $\mathsf{ob}^*; [\mathtt{Inst}] \subseteq \mathsf{ib}'^*$. So $\mathsf{ob}'; [\mathtt{Inst}] = \mathsf{ob}; [\mathtt{Inst}] \cup ((\mathsf{ob}^*; [\mathtt{Inst}]); \mathsf{ib}')^+; (\mathsf{ob}^*; [\mathtt{Inst}]) \subseteq \mathsf{ib}' \cup (\mathsf{ib}'^*; \mathsf{ib}')^+; \mathsf{ib}'^* \subseteq \mathsf{ib}'$.

Once ib is a total relation on Event, we can similarly freely extend ob into a total relation.

We use Theorem 11 above to extend ib and ob into total relations IB and OB.

Since $(\mathsf{Event}, \mathsf{po})$ is derived from P, by Appendix D.2 we have that for all $t \in \mathsf{Tid}$ there are s_t and G_t such that $G_t \in G^t(s_t)$, $\mathsf{P}(t) \rightarrowtail s_t$ and $(\mathsf{Event}, \mathsf{po}) = G_{init}; (\|_{t \in \mathsf{Tid}}\ G_t)$. We consider each premise of the form $C \rightarrowtail s$, where C is a primitive command, to generate new events and annotated labels.

- If $s = r \in \mathtt{lR}$, from well-formedness conditions, there is w such that $(w, r) \in \mathsf{rf}$ and $\mathsf{eq}_{\mathtt{loc\&v}}(r, w)$. We create an annotated label $\mathtt{lR}\langle r, w\rangle$.
- If $s = u, s'$ where $u \in \mathtt{CAS}$, from well-formedness conditions, there is w such that $(w, u) \in \mathsf{rf}$ and $\mathsf{eq}_{\mathtt{loc\&v}}(u, w)$. We create an annotated label $\mathtt{CAS}\langle u, w\rangle$, then process s'.
- If $s = f, r, s'$ where $f \in \mathtt{F}$, $r \in \mathtt{lR}$, and $w \in \mathtt{lW}$, from well-formedness conditions, there is w' such that $(w', r) \in \mathsf{rf}$ and $\mathsf{eq}_{\mathtt{loc\&v}}(r, w')$. We create annotated labels $\mathtt{F}\langle f\rangle$, $\mathtt{lR}\langle r, w'\rangle$, $\mathtt{lW}\langle w\rangle$ and $\mathsf{B}\langle w\rangle$, then process s'.
- If $s = w \in \mathtt{lW}$, we create annotated labels $\mathtt{lW}\langle w\rangle$ and $\mathsf{B}\langle w\rangle$.
- If $s = f \in \mathtt{F}$, we create annotated labels $\mathtt{F}\langle f\rangle$.

- If $s = r, w$ where $r \in \texttt{nlR}$ and $w \in \texttt{nrW}$, we create two events $a \in \texttt{Put}$ and $e \in \texttt{nrEX}$, and the annotated labels $\texttt{Push}\langle a\rangle$, $\texttt{NIC}\langle a\rangle$, $\texttt{nlR}\langle r, w', a, w\rangle$ (where $(w', r) \in \mathsf{rf}$), $\texttt{nrW}\langle w, e\rangle$, $\mathsf{B}\langle w\rangle$, and $\mathsf{CN}\langle e\rangle$. If there is p such that $(w, p) \in \mathsf{pf}$, we also create an annotated label $\texttt{P}\langle p, e\rangle$. To simplify later definition, we also extend po such that the event a is placed just before r, and e just after w. I.e., let $\mathsf{po}' = \mathsf{po} \cup \{(e', a) \mid (e', r) \in \mathsf{po}\} \cup \{(a, e') \mid (r, e') \in \mathsf{po}^*\}$ and redefine $\mathsf{po} = \mathsf{po}' \cup \{(e', e) \mid (e', w) \in \mathsf{po}'^*\} \cup \{(e, e') \mid (w, e') \in \mathsf{po}'\}$.
  Note: from well-formedness conditions, every $\texttt{nlR}$ and every $\texttt{nrW}$ are part of such a pair.
- If $s = r, w$ where $r \in \texttt{nrR}$ and $w \in \texttt{nlW}$, we similarly create $a \in \texttt{Get}$, $e \in \texttt{nlEX}$, $\texttt{Push}\langle a\rangle$, $\texttt{NIC}\langle a\rangle$, $\texttt{nrR}\langle\ldots\rangle$, $\texttt{nlW}\langle\ldots\rangle$, $\mathsf{B}\langle\ldots\rangle$, and potentially $\texttt{P}\langle\ldots\rangle$.
- If $s = r, w$ where $r \in \texttt{narR}$ and $w \in \texttt{nlW}$, we have C of the form $z := \texttt{RCAS}(\overline{x}, e, e')$, so we use the values $[\![e]\!]$ and $[\![e']\!]$ to create $a \in \texttt{RCAS}$, $\texttt{Push}\langle a\rangle$, $\texttt{NIC}\langle a\rangle$, $\texttt{naF}\langle\ldots\rangle$, $\texttt{nlW}\langle\ldots\rangle$, $\mathsf{B}\langle\ldots\rangle$, and potentially $\texttt{P}\langle\ldots\rangle$.
- If $s = r, w_1, w_2$ where $r \in \texttt{narR}$, $w_1 \in \texttt{narW}$, $w_2 \in \texttt{nlW}$, we have C either of the form $z := \texttt{RFAA}(\overline{x}, e)$ or $z := \texttt{RCAS}(\overline{x}, e_1, e_2)$, so we create $a \in \texttt{RFAA}$ or $a \in \texttt{RCAS}$ accordingly, and $\texttt{Push}\langle a\rangle$, $\texttt{NIC}\langle a\rangle$, $\texttt{narR}\langle\ldots\rangle$, $\texttt{narW}\langle w_1\rangle$, $\texttt{nlW}\langle w_2, \ldots\rangle$, $\mathsf{B}\langle w_1\rangle$, $\mathsf{B}\langle w_2\rangle$ and potentially $\texttt{P}\langle\ldots\rangle$.
- If $s = f \in \texttt{nF}$, we create the annotated labels $\texttt{Push}\langle f\rangle$, $\texttt{NIC}\langle f\rangle$, and $\texttt{nF}\langle f\rangle$.
- We ignore $s = p \in \texttt{P}$, as this is already handled by our earlier cases.

Then, we use IB and OB to reconstruct a partial path from these annotated labels. We define a path π_0 such that:

- $\pi_0 \in (\mathsf{ALabel} \setminus (\texttt{Push} \cup \texttt{NIC} \cup \mathsf{CN}))^*$
- $\mathsf{getI}\lambda(e_1, \pi_0) \prec_{\pi_0} \mathsf{getI}\lambda(e_2, \pi_0) \iff (e_1, e_2) \in \mathsf{IB}$
- $\mathsf{getO}\lambda(e_1, \pi_0) \prec_{\pi_0} \mathsf{getO}\lambda(e_2, \pi_0) \iff (e_1, e_2) \in \mathsf{OB}$
- $\forall w \in \{\texttt{lW}, \texttt{nlW}, \texttt{nrW}, \texttt{narW}\}$, $\mathsf{getI}\lambda(w, \pi_0) \prec_{\pi_0} \mathsf{getO}\lambda(w, \pi_0)$

This is possible from the properties of IB and OB. For pairs of annotated labels not ordered by IB or OB, we decide to order $\texttt{lW}\langle w\rangle/\texttt{nlW}\langle w, \_\rangle/\texttt{nrW}\langle w, \_\rangle/\texttt{narW}\langle w\rangle$ first and $\mathsf{B}\langle w\rangle$ last. Note that the annotated labels $\texttt{Push}\langle\ldots\rangle$, $\texttt{NIC}\langle\ldots\rangle$, and $\mathsf{CN}\langle\ldots\rangle$ not covered by IB/OB are not yet integrated in π_0.

Then we extend π_0 to add annotated labels not considered by the declarative semantics. We use the following extension function that introduces a new annotated label as early as possible after a set of dependencies.

$$\mathsf{extend}(\pi, \lambda, S) \triangleq \begin{cases} \pi_2 \cdot \lambda \cdot \lambda' \cdot \pi_1 & \text{if } \pi = \pi_2 \cdot \lambda' \cdot \pi_1 \wedge \lambda' \in S \wedge \pi_2 \cap S = \emptyset \\ \pi \cdot \lambda & \text{if } \pi \cap S = \emptyset \end{cases}$$

We define a new function to recover the first annotated label corresponding to an event:

$$E^{\text{ext}} \triangleq \mathsf{Event} \cup (\texttt{Get} \cup \texttt{Put} \cup \texttt{RCAS} \cup \texttt{RFAA} \cup \texttt{nlEX} \cup \texttt{nrEX})$$

$$\mathsf{getCPU} : E^{\text{ext}} \rightharpoonup \mathsf{ALabel}$$

$$\mathsf{getCPU}(e) \triangleq \begin{cases} \mathsf{getI}\lambda(e,\pi_0) & \text{if } e \in E^{\mathrm{cpu}} = \{\mathtt{lR},\mathtt{lW},\mathtt{CAS},\mathtt{F},\mathtt{P}\} \\ \mathtt{Push}\langle e\rangle & \text{if } e \in \{\mathtt{Put},\mathtt{Get},\mathtt{RCAS},\mathtt{RFAA},\mathtt{nF}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

And a similar function for events emptying a CPU buffer:

$$\mathsf{getTSO} : E^{\mathrm{ext}} \rightharpoonup \mathsf{ALabel}$$

$$\mathsf{getTSO}(e) \triangleq \begin{cases} \mathsf{B}\langle e\rangle & \text{if } e \in \mathtt{lW} \\ \mathtt{NIC}\langle e\rangle & \text{if } e \in \{\mathtt{Put},\mathtt{Get},\mathtt{RCAS},\mathtt{RFAA},\mathtt{nF}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Let us consider $(a_1,\ldots,a_n) = \mathsf{Event} \cap \{\mathtt{Put},\mathtt{Get},\mathtt{RCAS},\mathtt{RFAA},\mathtt{nF}\}$ in po order, i.e., if $i<j$ then $(a_j,a_i) \notin \mathsf{po}$. We extend π_0 successively until we get π_n:

- We introduce `Push` as early as possible:
  Let $\pi' = \mathsf{extend}(\pi_{i-1}, \mathtt{Push}\langle a_i\rangle, \{\mathsf{getCPU}(e) \mid (e,a_i) \in \mathsf{po}\})$
- We introduce `NIC` as early as possible:
  Let $\pi'' = \mathsf{extend}(\pi', \mathtt{NIC}\langle a_i\rangle, \{\mathtt{Push}\langle a_i\rangle\} \cup \{\mathsf{getTSO}(e) \mid (e,a_i) \in \mathsf{po}\})$
- If $a_i \in \mathtt{Put}$, there is $e_i \in \mathtt{nrEX}$ such that $\mathtt{nlR}\langle \_,\_,a_i,w\rangle \prec_{\pi_0} \mathtt{nrW}\langle w,e_i\rangle$. We also introduce CN: Let $S = \{\mathtt{nrW}\langle w,e_i\rangle\} \cup \{\mathtt{nlW}\langle \_,e\rangle \mid (e,e_i) \in \mathsf{po} \cap \mathtt{sqp}\} \cup \{\mathsf{CN}\langle e\rangle \mid (e,e_i) \in \mathsf{po} \cap \mathtt{sqp}\}$, we pose $\pi_i = \mathsf{extend}(\pi'', \mathsf{CN}\langle e_i\rangle, S)$.
  Otherwise, i.e. $a_i \notin \mathtt{Put}$, we simply have $\pi_i = \pi''$

Finally, $\pi = \pi_n$ is our path for an annotated semantics reduction. We clearly have $\mathsf{complete}(\pi)$ by definition. Our goal is then to prove that $\mathsf{wfp}(\pi)$ holds. It is composed of seven properties. Note that we already have the existence of the relevant annotated labels, and we need to show that the ordering constraints are respected.

nodup

$\mathsf{nodup}(\pi)$ directly comes from the definition of annotated labels. There is no conflict in event usage.

backComp

Here are a couple lemmas showing that the new annotated labels are not placed too late and do not disturb the expected ordering.

Lemma 4. *For all* $a \in \{\mathtt{Put},\mathtt{Get},\mathtt{RCAS},\mathtt{RFAA},\mathtt{nF}\}$ *and* $b \in \mathsf{Event}$*, if* $(a,b) \in \mathsf{po}^*$*, then* $\mathtt{Push}\langle a\rangle \prec_\pi \mathsf{getI}\lambda(b,\pi_0)$*.*

Proof. We take an arbitrary b, and proceed for a in po order, i.e., we can assume it holds for $e \in \{\mathtt{Put},\mathtt{Get},\mathtt{RCAS},\mathtt{RFAA},\mathtt{nF}\}$ such that $(e,a) \in \mathsf{po}$. By definition, $\mathtt{Push}\langle a\rangle$ comes from an extension $\pi'' = \mathsf{extend}(\pi', \mathtt{Push}\langle a\rangle, \{\mathsf{getCPU}(e) \mid (e,a) \in \mathsf{po}\})$ and has been placed either first—and the result is trivial—or just after some $\mathsf{getCPU}(e)$ with $(e,a) \in \mathsf{po}$. If $e \in \{\mathtt{Put},\mathtt{Get},\mathtt{RCAS},\mathtt{RFAA},\mathtt{nF}\}$, we have $\mathtt{Push}\langle e\rangle \prec_{\pi''} \mathtt{Push}\langle a\rangle \prec_{\pi''} \mathsf{getI}\lambda(b,\pi_0)$ by induction hypothesis. If $e \in E^{\mathrm{cpu}} = \{\mathtt{lR},\mathtt{lW},\mathtt{CAS},\mathtt{F},\mathtt{P}\}$, we have $\mathsf{getI}\lambda(e,\pi_0) \prec_{\pi''} \mathtt{Push}\langle a\rangle \prec_{\pi''} \mathsf{getI}\lambda(b,\pi_0)$ since $(e,b) \in \mathsf{ippo} \subseteq \mathsf{IB}$.

Lemma 5. $\forall a \in \{\texttt{Put}, \texttt{Get}, \texttt{RCAS}, \texttt{RFAA}, \texttt{nF}\}$, $\forall b \in \{\texttt{nF}, \texttt{nrR}, \texttt{nlR}, \texttt{narR}, \texttt{lW}\}$, *if* $(a, b) \in \mathsf{po}^*$, *then* $\texttt{NIC}\langle a\rangle \prec_\pi \mathsf{getO}\lambda(b, \pi_0)$.

Proof. We take an arbitrary $b \in \{\texttt{nF}, \texttt{nrR}, \texttt{nlR}, \texttt{narR}\}$, and proceed for a in po order, i.e., we can assume it holds for $e \in \{\texttt{Put}, \texttt{Get}, \texttt{RCAS}, \texttt{RFAA}, \texttt{nF}\}$ such that $(e, a) \in \mathsf{po}$. By definition, $\texttt{NIC}\langle a\rangle$ comes from an extension $\pi'' = \mathsf{extend}(\pi', \texttt{NIC}\langle a\rangle, S)$, with $S = \{\texttt{Push}\langle a\rangle\} \cup \{\mathsf{getTSO}(e) \mid (e, a) \in \mathsf{po}\}$, and has been placed just after some $\lambda \in S$.

- If $\lambda = \texttt{Push}\langle a\rangle$, then we have $\lambda \prec_{\pi''} \texttt{NIC}\langle a\rangle \prec_{\pi''} \mathsf{getO}\lambda(b, \pi_0)$ using Lemma 4 above, since $\mathsf{getI}\lambda(b, \pi_0) = \mathsf{getO}\lambda(b, \pi_0)$ or $\mathsf{getI}\lambda(b, \pi_0) \prec_{\pi''} \mathsf{getO}\lambda(b, \pi_0)$.
- If $\lambda = \mathsf{getTSO}\langle e\rangle = \texttt{NIC}\langle e\rangle$ for some $e \in \{\texttt{Put}, \texttt{Get}, \texttt{RCAS}, \texttt{RFAA}, \texttt{nF}\}$, then we have $\lambda \prec_{\pi''} \texttt{NIC}\langle a\rangle \prec_{\pi''} \mathsf{getO}\lambda(b, \pi_0)$ by induction hypothesis.
- If $\lambda = \mathsf{getTSO}\langle e\rangle = \mathsf{B}\langle e\rangle$ for some $e \in \texttt{lW}$, then we have $\mathsf{B}\langle e\rangle \prec_{\pi''} \texttt{NIC}\langle a\rangle \prec_{\pi''} \mathsf{getO}\lambda(b, \pi_0)$ since $(e, b) \in \mathsf{oppo} \subseteq \mathsf{OB}$.

Lemma 6. *Forall* w,e,p, *if* $\texttt{nrW}\langle w, e\rangle \in \pi$ *and* $\mathsf{P}\langle p, e\rangle \in \pi$, *then* $\mathsf{CN}\langle e\rangle \prec_\pi \mathsf{P}\langle p, e\rangle$.

Proof. Once again, we proceed for e in po order, i.e., we can assume the result holds for $e' \in \texttt{nrEX}$ such that $(e', e) \in \mathsf{po}$. $\mathsf{CN}\langle e\rangle$ is inserted in some operation $\pi'' = \mathsf{extend}(\pi', \mathsf{CN}\langle e\rangle, S)$, with $S = \{\texttt{nrW}\langle w, e\rangle\} \cup \{\texttt{nlW}\langle \_, e'\rangle \mid (e', e) \in \mathsf{po} \cap \texttt{sqp}\} \cup \{\mathsf{CN}\langle e'\rangle \mid (e', e) \in \mathsf{po} \cap \texttt{sqp}\}$. It is then placed just after some label $\lambda \in S$.

- If $\lambda = \texttt{nrW}\langle w, e\rangle$, we have $\lambda \prec_{\pi''} \mathsf{CN}\langle e\rangle \prec_{\pi''} \mathsf{P}\langle p, e\rangle$ because $(w, p) \in \mathsf{pf} \subseteq \mathsf{IB}$.
- If $\lambda = \mathsf{CN}\langle e'\rangle$ with $(e', e) \in \mathsf{po} \cap \texttt{sqp}$, then there is some w' such that $(w', w) \in \mathsf{po} \cap \texttt{sqp}$ and $\texttt{nrW}\langle w', e'\rangle \in \pi'$. From well-formedness condition number 1 (see Definition 19), there is some p' such that $(w', p') \in \mathsf{pf}$ and $(p', p) \in \mathsf{po}$. By induction hypothesis, we have $\mathsf{CN}\langle e'\rangle \prec_{\pi'} \mathsf{P}\langle p', e'\rangle$, and from $(p', p) \in \mathsf{IB}$ we have $\mathsf{P}\langle p', e'\rangle \prec_{\pi'} \mathsf{P}\langle p, e\rangle$. In the end, we have the result $\mathsf{CN}\langle e'\rangle \prec_{\pi''} \mathsf{CN}\langle e\rangle \prec_{\pi''} \mathsf{P}\langle p, e\rangle$.
- If $\lambda = \texttt{nlW}\langle w', e'\rangle$ with $(e', e) \in \mathsf{po} \cap \texttt{sqp}$, then we also have $(w', w) \in \mathsf{po} \cap \texttt{sqp}$, so from well-formedness condition number 1 (see Definition 19), there is some p' such that $(w', p') \in \mathsf{pf}$ and $(p', p) \in \mathsf{po}$. We have $\texttt{nlW}\langle w', e'\rangle \prec_{\pi''} \mathsf{CN}\langle e\rangle \prec_{\pi''} \mathsf{P}\langle p', e'\rangle \prec_{\pi''} \mathsf{P}\langle p, e\rangle$.

We can then check that we have $\mathsf{backComp}(\pi)$:

- $\texttt{lW}\langle w\rangle \prec_\pi \mathsf{B}\langle w\rangle$ comes from the third property when defining π_0; similarly for nlW, nrW and narW.
- $\texttt{Push}\langle a\rangle \prec_\pi \texttt{NIC}\langle a\rangle$ comes from the extension process.
- $\texttt{NIC}\langle f\rangle \prec_\pi \texttt{nF}\langle f\rangle$ comes from Lemma 5; similarly for $\texttt{NIC}\langle a\rangle \prec_\pi \texttt{nlR/nrR/naF/narR}\langle \ldots\rangle$.
- $\texttt{nlR}\langle r, w, a, w'\rangle \prec_\pi \texttt{nrW}\langle w', e\rangle$ comes from $(r, w') \in \mathsf{ippo} \subseteq \mathsf{IB}$; similarly for nrR/nlW, naF/nlW, narR/nlW and narR/narW.
- $\texttt{nrW}\langle w, e\rangle \prec_\pi \mathsf{CN}\langle e\rangle$ comes from the extension process
- $\texttt{nlW}\langle w, e\rangle \prec_\pi \mathsf{B}\langle w\rangle \prec_\pi \mathsf{P}\langle p, e\rangle$ comes from $(w, p) \in [\texttt{nlW}]; \mathsf{pf} \subseteq \mathsf{OB}$.
- $\mathsf{CN}\langle e\rangle \prec_\pi \mathsf{P}\langle p, e\rangle$ comes from Lemma 6.

Thus we have $\mathsf{backComp}(\pi)$.

bufFlushOrd

- $\texttt{lW}\langle w_1\rangle \prec_\pi \texttt{lW}\langle w_2\rangle \iff \mathsf{B}\langle w_1\rangle \prec_\pi \mathsf{B}\langle w_2\rangle$ when $t(w_1) = t(w_2)$ comes the fact that $[\texttt{lW}]; \mathsf{po}; [\texttt{lW}] \subseteq (\mathsf{IB} \cup \mathsf{OB})$, so both sides are true if and only if $(w_1, w_2) \in \mathsf{po}$; similarly for $\texttt{nlW}$ and $\texttt{nrW}/\texttt{narW}$ on the same queue pair.
- When $t(a_1) = t(a_2)$, $\texttt{Push}\langle a_1\rangle \prec_\pi \texttt{Push}\langle a_2\rangle \iff \texttt{NIC}\langle a_1\rangle \prec_\pi \texttt{NIC}\langle a_2\rangle \iff (a_1, a_2) \in \mathsf{po}$ from the definition of the extension process (to define π_n).
- For $a \in \{\texttt{Put}, \texttt{Get}, \texttt{nF}, \texttt{RCAS}, \texttt{RFAA}\}$, $w \in \texttt{lW}$, such that $t(a) = t(w)$:
  - If $(w, a) \in \mathsf{po}$, then $\texttt{lW}\langle w\rangle \prec_\pi \texttt{Push}\langle a\rangle$ and $\mathsf{B}\langle w\rangle \prec_\pi \texttt{NIC}\langle a\rangle$ from the definition of the extension process.
  - If $(a, w) \in \mathsf{po}$, then $\texttt{Push}\langle a\rangle \prec_\pi \texttt{lW}\langle w\rangle$ and $\texttt{NIC}\langle a\rangle \prec_\pi \mathsf{B}\langle w\rangle$ from Lemmas 4 and 5.
- When $t(w) = t(f)$, $\texttt{lW}\langle w\rangle \prec_\pi \texttt{F}\langle f\rangle$ implies $(w, f) \in \mathsf{po}$ (since $[\texttt{F}]; \mathsf{po}; [\texttt{lW}] \subseteq \mathsf{ippo} \subseteq \mathsf{IB}$), which implies $\mathsf{B}\langle w\rangle \prec_\pi \texttt{F}\langle f\rangle$ (since $[\texttt{lW}]; \mathsf{po}; [\texttt{F}] \subseteq \mathsf{oppo} \subseteq \mathsf{OB}$); similarly for $\texttt{CAS}$.
- If $w \in \texttt{nlW}$, $r \in \texttt{nlR}$, and $\mathsf{sameqp}(w, r)$, then from the definition of pre-executions (see condition 6 of Definition 18), either $(w, r) \in \mathsf{nfo}$ or $(r, w) \in \mathsf{nfo}$. If $\texttt{nlW}\langle w, \_\rangle \prec_\pi \texttt{nlR}\langle r, \_, \_, \_\rangle$, then $(r, w) \notin \mathsf{nfo}$ (since $\mathsf{nfo} \subseteq \mathsf{IB}$) and $(w, r) \in \mathsf{nfo}$. Thus, $\mathsf{B}\langle w\rangle \prec_\pi \texttt{nlR}\langle r, \_, \_, \_\rangle$ (since $\mathsf{nfo} \subseteq \mathsf{OB}$); similarly for $w \in \{nrW, narW\}$ and $r \in \{\texttt{nrR}, \texttt{narR}\}$.

Thus we have $\mathsf{bufFlushOrd}(\pi)$.

pollOrder

Lemma 7. *For all* $e_1, e_2 \in \{\texttt{nlEX}, \texttt{nrEX}\}$*, such that* $\mathsf{sameqp}(e_1, e_2)$*, let* $\lambda_1 \in \{\texttt{nlW}\langle \_, e_1\rangle, \mathsf{CN}\langle e_1\rangle\}$*,* $\lambda_2 \in \{\texttt{nlW}\langle \_, e_2\rangle, \mathsf{CN}\langle e_2\rangle\}$*, then* $(e_1, e_2) \in \mathsf{po} \iff \lambda_1 \prec_\pi \lambda_2$.

Proof. By symmetry, we only need to show $(e_1, e_2) \in \mathsf{po} \implies \lambda_1 \prec_\pi \lambda_2$. Once again, we proceed for e_1 in po order, i.e., we can assume the result holds for $e' \in \texttt{nEX}$ such that $(e', e_1) \in \mathsf{po}$.

- If $\lambda_1 = \texttt{nlW}\langle w_1, e_1\rangle$ and $\lambda_2 = \texttt{nlW}\langle w_2, e_2\rangle$, then $(e_1, e_2) \in \mathsf{po}$ implies $(w_1, w_2) \in (\mathsf{po} \cap \texttt{sqp})$, so $(w_1, w_2) \in \mathsf{ippo} \subseteq \mathsf{IB}$ and $\lambda_1 \prec_\pi \lambda_2$.
- If $\lambda_1 = \texttt{nlW}\langle w_1, e_1\rangle$ and $\lambda_2 = \mathsf{CN}\langle e_2\rangle$, then by definition of the extension process we have $\lambda_1 \prec_\pi \lambda_2$.
- If $\lambda_1 = \mathsf{CN}\langle e_1\rangle$ and $\lambda_2 = \texttt{nlW}\langle w_2, e_2\rangle$, then λ_1 is inserted in some operation $\pi'' = \mathsf{extend}(\pi', \mathsf{CN}\langle e_1\rangle, S)$, with $S = \{\texttt{nrW}\langle \_, e_1\rangle\} \cup \{\texttt{nlW}\langle \_, e'\rangle \mid (e', e_1) \in \mathsf{po} \cap \texttt{sqp}\} \cup \{\mathsf{CN}\langle e'\rangle \mid (e', e_1) \in \mathsf{po} \cap \texttt{sqp}\}$. It is then placed just after some label $\lambda \in S$.
  - If $\lambda = \texttt{nrW}\langle w_1, e_1\rangle$, we have $\lambda \prec_{\pi''} \lambda_1 \prec_{\pi''} \lambda_2$ because $(w_1, w_2) \in \mathsf{ippo} \subseteq \mathsf{IB}$.
  - If $\lambda = \mathsf{CN}\langle e'\rangle$ or $\lambda = \texttt{nlW}\langle \_, e'\rangle$, with $(e', e_1) \in \mathsf{po} \cap \texttt{sqp}$, then by induction hypothesis $\lambda \prec_{\pi''} \lambda_1 \prec_{\pi''} \lambda_2$.
- If $\lambda_1 = \mathsf{CN}\langle e_1\rangle$ and $\lambda_2 = \mathsf{CN}\langle e_2\rangle$, then by definition of the extension process we have $\lambda_1 \prec_\pi \lambda_2$.

Let us assume we have $e_1, e_2, p_2, \lambda_1, \lambda_2$ such that $\mathsf{sameqp}(e_1, e_2)$, $\lambda_1 \in \{\texttt{nlW}\langle \_, e_1\rangle, \mathsf{CN}\langle e_1\rangle\}$, $\lambda_2 \in \{\texttt{nlW}\langle \_, e_2\rangle, \mathsf{CN}\langle e_2\rangle\}$, $\lambda_1 \prec_\pi \lambda_2$, and $\texttt{P}\langle p_2, e_2\rangle \in \pi$.

From the creation of the events e_1 and e_2, there is some $w_1, w_2 \in \{\texttt{nlW}, \texttt{nrW}\}$ such that $(w_i, e_i) \in \mathsf{po}|_{\mathrm{imm}}$. From Lemma 7, we have $(e_1, e_2) \in \mathsf{po}$ and thus

$(w_1, w_2) \in (\mathsf{po} \cap \mathtt{sqp})$. By definition, we also have $(w_2, p_2) \in \mathsf{pf}$. From well-formedness condition number 1 (see Definition 19), there is some p_1 such that $(w_1, p_1) \in \mathsf{pf}$ and $(p_1, p_2) \in \mathsf{po}$. Thus we have $\mathtt{P}\langle p_1, e_1\rangle \prec_\pi \mathtt{P}\langle p_2, e_2\rangle$ as required to prove $\mathsf{pollOrder}(\pi)$.

nicActOrder

Let a_1 and a_2 such that $\mathtt{NIC}\langle a_1\rangle \prec_\pi \mathtt{NIC}\langle a_2\rangle$ and $\mathsf{sameqp}(a_1, a_2)$. From the definition of the extension process, we have $(a_1, a_2) \in \mathsf{po}$.

- If $a_1 \in \mathtt{nF}$ or $a_2 \in \mathtt{nF}$, then most of the required results hold by definition of ippo. The only exception is $\mathsf{CN}\langle e\rangle \prec_\pi \mathtt{nF}\langle a_2\rangle$ which holds (by induction on e in po order) because all the dependencies of $\mathsf{CN}\langle e\rangle$ are before $\mathtt{nF}\langle a_2\rangle$ by ippo.
- If $(a_1 \in \mathtt{Get} \wedge a_2 \in \mathtt{Get})$, the result holds by ippo.
- If $(a_1 \in \mathtt{Get} \wedge a_2 \in \mathtt{Put})$, the result holds by Lemma 7.
- If $(a_1 \in \mathtt{Get} \wedge a_2 \in \mathtt{RCAS} \cup \mathtt{RFAA})$, the results hold by ippo.
- If $(a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{Get})$, the first result holds by ippo, the second by Lemma 7.
- If $(a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{Put})$, the first two results hold by ippo, the last one by Lemma 7.
- If $(a_1 \in \mathtt{Put} \wedge a_2 \in \mathtt{RCAS} \cup \mathtt{RFAA})$, the results hold by ippo.
- If $(a_1 \in \mathtt{RCAS} \cup \mathtt{RFAA} \wedge a_2 \in \mathtt{Get})$, the results hold by ippo.
- If $(a_1 \in \mathtt{RCAS} \cup \mathtt{RFAA} \wedge a_2 \in \mathtt{Put})$, the first result holds by ippo, the latter by Lemma 7.
- If $(a_1, a_2 \in \mathtt{RCAS} \cup \mathtt{RFAA})$, the first result holds by ippo, the latter by Lemma 7.

Thus we have $\mathsf{nicActOrder}(\pi)$.

nicAtomicity

For every $a_1, a_2 \in \mathtt{rRMW}$ where $\overline{n}(a_1) = \overline{n}(a_2)$, if $\mathtt{narR}\langle r_1, a_1, \_, \_, w\rangle \prec_\pi \lambda_r$ where $\lambda_r \in \{\mathtt{naF}\langle r_2, \_, a_2, \_\rangle, \mathtt{narR}\langle r_2, \_, a_2, \_, \_\rangle\}$, then from the extension process we have $(r_1, w) \in \mathsf{po}|_{\mathrm{imm}}$, and $w \in \mathtt{narW}$, so $(w, r_1) \in \mathsf{ar}$. Then we need to show that $(r_1, r_2) \in \mathsf{rao}$. Suppose, for contradiction, that $(r_1, r_2) \notin \mathsf{rao}$. By definition of rao, for each node n, rao_n is a total order on $\{e \in \mathtt{narR} \mid \overline{n}(e) = n\}$. Thus we have either $(r_1, r_2) \in \mathsf{rao}$ or $(r_2, r_1) \in \mathsf{rao}$, and by assumption the prior is not the case so $(r_2, r_1) \in \mathsf{rao} \subseteq \mathsf{OB}$. However, since $\mathtt{narR}\langle r_1, \ldots\rangle \prec_\pi \lambda_r$, we have $(r_1, r_2) \in \mathsf{OB}$, which is a contradiction, as OB is irreflexive. Therefore reject our original assumption. Thus $(r_1, r_2) \in \mathsf{rao}$, then we have $(w, r_2) \in \mathsf{ar}; \mathsf{rao} \subseteq \mathsf{OB}$, so $\mathsf{B}\langle w\rangle \prec_\pi \lambda_r$. Thus we have $\mathsf{nicAtomicity}(\pi)$.

wfrd

Let us assume we have $\pi = \pi_4 \cdot \lambda_r \cdot \pi_3$, with $\lambda_r \in \{\mathtt{lR}\langle r, w\rangle, \mathtt{CAS}\langle r, w\rangle, \mathtt{nlR}\langle r, w, \_, \_\rangle, \mathtt{nrR}\langle r, w, \_, \_\rangle, \mathtt{naF}\langle r, w, \_, \_\rangle, \mathtt{narR}\langle r, w, \_, \_, \_\rangle\}$. In all cases we have $(w, r) \in \mathsf{rf}$. Another important fact is that $\forall w', (w, w') \in \mathsf{mo} \implies (r, w') \in \mathsf{rb}$.

- If $\lambda_r = \mathtt{lR}\langle r, w\rangle$, we need to show $\mathsf{wfrdCPU}(r, w, \pi_3)$.
  - If $w = init_{\mathtt{loc}(w)}$, then we need to check that $\{\mathsf{B}\langle w'\rangle, \mathtt{CAS}\langle w', \_\rangle \in \pi_3 \mid \mathtt{loc}(w') = \mathtt{loc}(r)\} = \emptyset$ and $\{\mathtt{lW}\langle w''\rangle \in \pi_3 \mid \mathtt{loc}(w'') = \mathtt{loc}(r) \wedge t(w'') = t(r)\} = \emptyset$. For the first, such a w' would imply $(r, w') \in \mathsf{rb} \subseteq \mathsf{OB}$, which contradicts the ordering with λ_r. For the second, such an w'' would imply $(r, w'') \in \mathsf{rb_i} \subseteq \mathsf{IB}$, and $\lambda_r \prec_\pi \mathtt{lW}\langle w''\rangle$ which similarly contradicts the ordering with λ_r.

- If $w \in \texttt{lW}$, $t(w) = t(r)$, and $\mathsf{B}\langle w\rangle \notin \pi_3$. From $(w,r) \in \mathsf{rf_i} \subseteq \mathsf{IB}$, we have $\lambda_w = \texttt{lW}\langle w\rangle \prec_\pi \lambda_r$, i.e., $\pi_3 = \pi_2 \cdot \lambda_w \cdot \pi_1$. We need to show that $\{\texttt{lW}\langle w'\rangle \in \pi_2 \mid \texttt{loc}(w') = \texttt{loc}(r) \wedge t(w') = t(r)\} = \emptyset$. Such a w' would imply $(w,w') \in \mathsf{po}$ (from $[\texttt{lW}];\mathsf{po};[\texttt{lW}] \subseteq \mathsf{ippo} \subseteq \mathsf{IB}$, and the execution graph forcing either $(w,w') \in \mathsf{po}$ or $(w',w) \in \mathsf{po}$), $(w,w') \in \mathsf{mo}$ (from $[\texttt{lW}];\mathsf{po};[\texttt{lW}] \subseteq \mathsf{oppo} \subseteq \mathsf{OB}$, and well-formedness conditions forcing either $(w,w') \in \mathsf{mo}$ or $(w',w) \in \mathsf{mo}$), and $(r,w') \in \mathsf{rb_i} \subseteq \mathsf{IB}$ would contradicts the ordering with λ_r.
- Else we have $\lambda_w \in \pi_3$, with $\lambda_w \in \{\mathsf{B}\langle w\rangle, \texttt{CAS}\langle w,\_\rangle\}$. If $w \in \texttt{lW}$ and $t(w) = t(r)$, this is the remaining subcase, else it comes from $(w,r) \in \mathsf{rf_e} \subseteq \mathsf{OB}$. Thus we have $\pi_3 = \pi_2 \cdot \lambda_w \cdot \pi_1$, and we need to check two properties. First, we check that $\{\mathsf{B}\langle w'\rangle, \texttt{CAS}\langle w',\_\rangle \in \pi_2 \mid \texttt{loc}(w') = \texttt{loc}(r)\} = \emptyset$. It holds because such a w' would again imply $(r,w') \in \mathsf{rb} \subseteq \mathsf{OB}$ and contradict the ordering with λ_r. Second, we check that $\left\{ w' \;\middle|\; \begin{array}{l} \texttt{lW}\langle w'\rangle \in \pi_3 \wedge \mathsf{B}\langle w'\rangle \notin \pi_3 \wedge \\ \texttt{loc}(w') = \texttt{loc}(r) \wedge t(w') = t(r) \end{array} \right\} = \emptyset$. It holds because such a w' would again imply $(w,w') \in \mathsf{mo}$, $(r,w') \in \mathsf{rb_i} \subseteq \mathsf{IB}$ and contradict the ordering with λ_r.
- If $\lambda_r = \texttt{CAS}\langle r,w\rangle$, we similarly check that $\mathsf{wfrdCPU}(r,w,\pi_3)$ holds. The difference is that cases that previously contradicted ($\mathsf{rb_i} \subseteq \mathsf{IB}$) now contradict $\mathsf{bufFlushOrd}(\pi)$ that forces the buffer of $t(r)$ to be empty when performing λ_r.
- If $\lambda_r = \texttt{nlR}\langle r,w,\_,\_\rangle$, we need to show $\mathsf{wfrdNIC}(r,w,\pi_3)$.
  - If $w = \mathit{init}_{\texttt{loc}(w)}$, then we need to check that $\{\mathsf{B}\langle w'\rangle, \texttt{CAS}\langle w',\_\rangle \in \pi_3 \mid \texttt{loc}(w') = \texttt{loc}(r)\} = \emptyset$. Such a w' would imply $(r,w') \in \mathsf{rb} \subseteq \mathsf{OB}$, which contradicts the ordering with λ_r.
  - Else we have $\lambda_w \in \pi_3$, with $\lambda_w \in \{\mathsf{B}\langle w\rangle, \texttt{CAS}\langle w,\_\rangle\}$. This comes from $(w,r) \in \mathsf{rf_e} \subseteq \mathsf{OB}$. Thus we have $\pi_3 = \pi_2 \cdot \lambda_w \cdot \pi_1$, and we need to check that $\{\mathsf{B}\langle w'\rangle, \texttt{CAS}\langle w',\_\rangle \in \pi_2 \mid \texttt{loc}(w') = \texttt{loc}(r)\} = \emptyset$. It holds because such a w' would again imply $(r,w') \in \mathsf{rb} \subseteq \mathsf{OB}$ and contradict the ordering with λ_r.
- If $\lambda_r = \texttt{nrR}\langle r,w,\_,\_\rangle$, $\texttt{naF}\langle r,w,\_,\_\rangle$ or $\texttt{narR}\langle r,w,\_,\_,\_\rangle$, we similarly check that $\mathsf{wfrdNIC}(r,w,\pi_3)$ for the same reasons.

Thus we have $\mathsf{wfrd}(\pi)$.

Theorem 12. *Let G be a well-formed consistent execution graph generated from a program P. Let π be the path obtained from G by the process defined above. Then there is M', QP' (such that forall $t, \overline{n}$ we have $\mathsf{QP}'(t)(\overline{n}) = \langle \varepsilon, \varepsilon, \texttt{nEX}^*\rangle$), and an equivalent path π' (producing the same outcome as π) such that $\mathsf{P}, \mathsf{M}_0, \mathsf{B}_0, \mathsf{A}_0, \mathsf{QP}_0, \varepsilon \Rightarrow^* (\lambda t.\texttt{skip}), \mathsf{M}', \mathsf{B}_0, \mathsf{A}_0, \mathsf{QP}', \pi'$.*

Proof. From above, we have $\mathsf{wf}(\pi)$. This shows that the program configuration can perform the events described by the annotated labels of π. The remaining part of the proof is simply to check that the command rewritings used when deriving the execution graph from P (see Fig. 23) can be used as $\mathcal{E}$ transitions in the annotated semantics for P, which follows from the definitions.

D.7 Operational Semantics and Annotated Semantics

We define forgetful functions from annotated configurations to operational configurations. For memories, we replace the write event by the value written. For labels within annotated configurations, we drop some arguments to recover the data structure of the operational semantics.

$$[\![\cdot]\!]_{\mathrm{M}} : \mathsf{AMem} \to \mathsf{Mem}$$
$$[\![\mathsf{M}]\!]_{\mathrm{M}} \triangleq \lambda x.v_{\mathrm{w}}(\mathsf{M}(x))$$

$$[\![\cdot]\!]_{\mathrm{op}} : E^{\mathrm{ext}} \rightharpoonup \left\{ \begin{array}{l} y^{\overline{n}} := x^n, y^{\overline{n}} := v, \mathtt{ack_p}, x^n := y^{\overline{n}}, x^n := v, \\ x := \mathtt{RCAS}(y^n, v, v'), x := \mathtt{RFAA}(y^n, v), \mathtt{cn}, \mathtt{rfence}(\overline{n}) \end{array} \right\}$$

$$\begin{aligned}
[\![\mathtt{lW}(x, v_{\mathrm{w}})]\!]_{\mathrm{op}} &\triangleq x := v_{\mathrm{w}} \\
[\![\mathtt{nrW}(\overline{y}, v_{\mathrm{r}})]\!]_{\mathrm{op}} &\triangleq \overline{y} := v_{\mathrm{r}} \\
[\![\mathtt{nlW}(x, v_{\mathrm{w}}, \overline{n})]\!]_{\mathrm{op}} &\triangleq x := v_{\mathrm{w}} \\
[\![\mathtt{nF}(\overline{n})]\!]_{\mathrm{op}} &\triangleq \mathtt{rfence}(\overline{n}) \\
[\![\mathtt{Put}(\overline{y}, x)]\!]_{\mathrm{op}} &\triangleq \overline{y} := x \\
[\![\mathtt{Get}(x, \overline{y})]\!]_{\mathrm{op}} &\triangleq x := \overline{y} \\
[\![\mathtt{RCAS}(z, \overline{x}, v, v')]\!]_{\mathrm{op}} &\triangleq z := \mathtt{RCAS}(\overline{x}, v, v') \\
[\![\mathtt{RFAA}(z, \overline{x}, v)]\!]_{\mathrm{op}} &\triangleq z := \mathtt{RFAA}(\overline{x}, v) \\
[\![\mathtt{nlEX}(\overline{n})]\!]_{\mathrm{op}} &\triangleq \mathtt{cn} \\
[\![\mathtt{nrEX}(\overline{n})]\!]_{\mathrm{op}} &\triangleq \mathtt{ack_p}
\end{aligned}$$

$$\begin{aligned}
[\![\mathtt{F}]\!]_{\mathrm{op}} &\text{ is undefined} \\
[\![\mathtt{P}(\ldots)]\!]_{\mathrm{op}} &\text{ is undefined} \\
[\![\mathtt{lR}(\ldots)]\!]_{\mathrm{op}} &\text{ is undefined} \\
[\![\mathtt{CAS}(\ldots)]\!]_{\mathrm{op}} &\text{ is undefined} \\
[\![\mathtt{nlR}(\ldots)]\!]_{\mathrm{op}} &\text{ is undefined} \\
[\![\mathtt{nrR}(\ldots)]\!]_{\mathrm{op}} &\text{ is undefined} \\
[\![\mathtt{naF}(\ldots)]\!]_{\mathrm{op}} &\text{ is undefined} \\
[\![\mathtt{narR}(\ldots)]\!]_{\mathrm{op}} &\text{ is undefined} \\
[\![\mathtt{narW}(\ldots)]\!]_{\mathrm{op}} &\text{ is undefined}
\end{aligned}$$

$$[\![\cdot]\!]_{\mathrm{opl}} : E^{\mathrm{ext}} \rightharpoonup \left\{ \begin{array}{l} y^{\overline{n}} := x^n, y^{\overline{n}} := v, x^n := y^{\overline{n}}, x^n := v, \\ x := \mathtt{RCAS}(y^n, v, v'), x := \mathtt{RFAA}(y^n, v), \mathtt{cn}, \mathtt{rfence}(\overline{n}) \end{array} \right\}$$

$$[\![l]\!]_{\mathrm{opl}} = \begin{cases} \mathtt{cn} & \text{if } l = \mathtt{nrEX}(\overline{n}) \\ [\![l]\!]_{\mathrm{op}} & \text{otherwise} \end{cases}$$

The labels that cannot appear in a well-formed annotated configuration are not mapped. For put operations, the operational semantics uses both ($\mathtt{ack_p}$) and ($\mathtt{cn}$) while the annotated semantics uses the label $\mathtt{nrEX}$, so the mapping is different for labels in $\mathbf{wb_L}$.

$[\![\cdot]\!]_{\mathrm{op}}$ and $[\![\cdot]\!]_{\mathrm{opl}}$ are extended to lists in an obvious way.

We then extend this to configurations as expected. We overload notations to simplify the formulas.

For $\mathsf{qp} = \langle \mathbf{pipe}, \mathbf{wb_R}, \mathbf{wb_L} \rangle \in \mathsf{AQPair}$, we define $[\![\mathsf{qp}]\!] \triangleq \langle [\![\mathbf{pipe}]\!]_{\mathrm{op}}, [\![\mathbf{wb_R}]\!]_{\mathrm{op}}, [\![\mathbf{wb_L}]\!]_{\mathrm{opl}} \rangle$.

For $\mathsf{QP} \in \mathsf{AQPMap}$, we define $[\![\mathsf{QP}]\!] \triangleq \lambda t.\lambda \overline{n}.[\![\mathsf{QP}(t)(\overline{n})]\!]$.

For $\mathsf{B} \in \mathsf{ASBMap}$, we define $[\![\mathsf{B}]\!] \triangleq \lambda t.[\![\mathsf{B}(t)]\!]_{\mathrm{op}}$.

Theorem 13. *For all* $\mathsf{P}, \mathsf{P}' \in \mathsf{Prog}$, $\mathsf{M}, \mathsf{M}' \in \mathsf{AMem}$, $\mathsf{B}, \mathsf{B}' \in \mathsf{ASBMap}$, $\mathsf{A}, \mathsf{A}' \in \mathsf{RAMap}$, $\mathsf{QP}, \mathsf{QP}' \in \mathsf{AQPMap}$, $\pi, \pi' \in \mathsf{Path}$, *if* $\mathsf{P}, \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi \Rightarrow \mathsf{P}', \mathsf{M}', \mathsf{B}', \mathsf{A}', \mathsf{QP}', \pi'$ *and* $\mathsf{wf}(\mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi)$, *then* $\mathsf{P}, [\![\mathsf{M}]\!]_{\mathrm{M}}, [\![\mathsf{B}]\!], \mathsf{A}, [\![\mathsf{QP}]\!] \Rightarrow \mathsf{P}', [\![\mathsf{M}']\!]_{\mathrm{M}}, [\![\mathsf{B}']\!], \mathsf{A}', [\![\mathsf{QP}']\!]$.

Proof. By straightforward induction on $\Rightarrow$.

Theorem 14. *For all* $\mathsf{M} \in \mathsf{AMem}$, $\mathsf{M}'' \in \mathsf{Mem}$, $\mathsf{B} \in \mathsf{ASBMap}$, $\mathsf{B}'' \in \mathsf{SBMap}$, $\mathsf{A}, \mathsf{A}' \in \mathsf{RAMap}$, $\mathsf{QP} \in \mathsf{AQPMap}$, $\mathsf{QP}'' \in \mathsf{QPMap}$, *and* $\pi \in \mathsf{Path}$, *if* $\mathsf{P}, [\![\mathsf{M}]\!]_{\mathrm{M}}, [\![\mathsf{B}]\!], \mathsf{A}, [\![\mathsf{QP}]\!] \Rightarrow \mathsf{P}', \mathsf{M}'', \mathsf{B}'', \mathsf{A}', \mathsf{QP}''$ *and* $\mathsf{wf}(\mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi)$, *then there exists* $\mathsf{M}' \in \mathsf{AMem}$, $\mathsf{B}' \in \mathsf{ASBMap}$, $\mathsf{QP}' \in \mathsf{AQPMap}$, *and* $\pi' \in \mathsf{Path}$ *such that* $[\![\mathsf{M}']\!]_{\mathrm{M}} = \mathsf{M}''$, $[\![\mathsf{B}']\!] = \mathsf{B}''$, $[\![\mathsf{QP}']\!] = \mathsf{QP}''$, *and* $\mathsf{P}, \mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi \Rightarrow \mathsf{P}', \mathsf{M}', \mathsf{B}', \mathsf{A}', \mathsf{QP}', \pi'$.

Proof. By straightforward induction on $\Rightarrow$. In some cases, the reduction enforces a specific annotated label λ and we have $\pi' = \lambda \cdot \pi$; we then need $\mathsf{wf}(\mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi)$ to check that λ is fresh enough for π.

Theorem 15 (Operational and Annotated Semantics Equivalence). *For all program* P.

- $[\![\mathsf{M}_0]\!]_{\mathrm{M}}$, $[\![\mathsf{B}_0]\!]$, A_0, *and* $[\![\mathsf{QP}_0]\!]$ *are the initialisation for the operational semantics;*
- *If* $\mathsf{P}, \mathsf{M}_0, \mathsf{B}_0, \mathsf{A}_0, \mathsf{QP}_0, \varepsilon \Rightarrow^* \mathsf{P}', \mathsf{M}', \mathsf{B}', \mathsf{A}', \mathsf{QP}', \pi'$ *then* $\mathsf{P}, [\![\mathsf{M}_0]\!]_{\mathrm{M}}, [\![\mathsf{B}_0]\!], \mathsf{A}_0, [\![\mathsf{QP}_0]\!] \Rightarrow^* \mathsf{P}', [\![\mathsf{M}']\!]_{\mathrm{M}}, [\![\mathsf{B}']\!], \mathsf{A}', [\![\mathsf{QP}']\!]$
- *If* $\mathsf{P}, [\![\mathsf{M}_0]\!]_{\mathrm{M}}, [\![\mathsf{B}_0]\!], \mathsf{A}_0, [\![\mathsf{QP}_0]\!] \Rightarrow^* \mathsf{P}', \mathsf{M}'', \mathsf{B}'', \mathsf{A}', \mathsf{QP}''$ *then there exists* $\mathsf{M}' \in \mathsf{AMem}$, $\mathsf{B}' \in \mathsf{ASBMap}$, $\mathsf{QP}' \in \mathsf{AQPMap}$, *and* $\pi' \in \mathsf{Path}$ *such that* $[\![\mathsf{M}']\!]_{\mathrm{M}} = \mathsf{M}''$, $[\![\mathsf{B}']\!] = \mathsf{B}''$, $[\![\mathsf{QP}']\!] = \mathsf{QP}''$, *and* $\mathsf{P}, \mathsf{M}_0, \mathsf{B}_0, \mathsf{A}_0, \mathsf{QP}_0, \varepsilon \Rightarrow^* \mathsf{P}', \mathsf{M}', \mathsf{B}', \mathsf{A}', \mathsf{QP}', \pi'$.

Proof. The first point comes from unfolding the definitions. The other two are proved by straightforward induction on $\Rightarrow^*$ and using Theorems 13 and 14. The condition $\mathsf{wf}(\mathsf{M}, \mathsf{B}, \mathsf{A}, \mathsf{QP}, \pi)$ is obtained by applying Theorem 8 when needed.