Sufficient Conditions for Robustness of RDMA Programs

Guillaume Ambal[1], Ori Lahav[2], and Azalea Raad[1]

[1] Imperial College London, UK
{g.ambal,azalea.raad}@imperial.ac.uk
[2] Tel Aviv University, Israel
orilahav@tau.ac.il

Abstract. Remote Direct Memory Access (RDMA) is a modern technology enabling high-performance inter-node communication. Despite its widespread adoption, theoretical understanding of permissible behaviours remains limited, as RDMA follows a very weak memory model. This paper addresses the challenge of establishing sufficient conditions for RDMA robustness. We introduce a set of straightforward criteria that, when met, guarantee sequential consistency and mitigate potential issues arising from weak memory behaviours in RDMA applications. Notably, when restricted to a tree topology, these conditions become even more relaxed, significantly reducing the need for synchronisation primitives. This work provides developers with practical guidelines to ensure the reliability and correctness of their RDMA-based systems.

Keywords: RDMA · Robustness · Weak Memory Models

1 Introduction

Remote Direct Memory Access (RDMA) is a modern technology that enables a machine to have *direct* read/write access to the memory of another machine over a network, bypassing the operating systems on both ends. This allows such direct memory accesses (reads/writes) to be performed with far fewer CPU cycles, leading to high-throughput, low-latency networking, which is especially useful in massively parallel computer clusters (e.g. data centres). RDMA has achieved widespread adoption as of 2018 [69], thanks to efficient implementations available at comparable cost to traditional infrastructures (e.g. TCP/IP sockets) [32], with several RDMA technologies such as *InfiniBand* and *RDMA over Converged Ethernet* (RoCE) readily available.

RDMA networks directly interact with the hardware through read (get) and write (put) operations on remote memory. As a result, programming RDMA systems is conceptually similar to shared memory systems of existing hardware architectures (e.g. Intel-x86 or ARM). A key difference, however, is that on encountering a remote operation, the CPU forwards it onto the *network interface card* (NIC), which subsequently handles the remote operation without further CPU involvement.

The performance gains of RDMA, as well as its wide range of implementations, have led to a surge of RDMA research [4,73,71,27]. RDMA networks exhibit different degrees of concurrency, depending on whether the concurrent threads reside on different nodes (machines) over the network (inter-node concurrency) or on the same node (intra-thread concurrency). To understand the behaviour of RDMA programs and their various notions of concurrency, Ambal et al. [10] recently developed RDMA$^{\text{TSO}}$, a formal semantics of RDMA programs where each node comprises an Intel-x86 CPU and thus intra-node-inter-thread concurrency is governed by the TSO (total store ordering) model [68].

As the real power of RDMA networks is their ability to run parallel programs over different nodes, writing efficient RDMA programs hinges on utilising inter-node concurrency. However, writing such programs *correctly* is far from straightforward. A key challenge is that local operations (accessing the local memory of the executing node) are handled by the CPU, while remote operations (accessing remote memory on other nodes) are handled by the NIC independently and in *parallel* to CPU operations. Hence, operations in the same thread may not be executed in the intended (program) order, leading to surprising outcomes. As Ambal et al. [10] note, this can result in counter-intuitive behaviours even in the case of *sequential* programs comprising a single thread. This is in stark contrast to *all* previously existing concurrency models (be they of CPU architectures or programming languages), where sequential programs do behave sequentially.

The permissive nature of RDMA semantics requires developers to carefully consider potential instruction reorderings. Reasoning about concurrent programs and ensuring proper synchronisation between threads is inherently complex, even without instruction reordering. Accounting for instruction reorderings adds another layer of complexity to this challenge.

As such, we should ideally enable reasoning about RDMA programs under a simpler, more intuitive model such as *sequential consistency* (SC) [43], where no instruction reordering is allowed, and thus instructions in each thread always execute in order. To this end, a common approach to simplify reasoning is to ensure *robustness*. A program P is *robust* under a consistency model CM, if its set of possible behaviours under CM coincide with those of its behaviours under SC; i.e. P is robust under CM if it exhibits no non-SC behaviours. If a program is robust under CM, then we can simply reason about it under SC, without considering the complexities of CM.

Contributions. In this paper, we close this gap and simplify reasoning about RDMA program through robustness. To simplify our presentation and not distract the reader from the RDMA complexities by the *orthogonal* intricacies of CPU concurrency, we first present RDMA$^{\text{SC}}$, a simplification of the RDMA$^{\text{TSO}}$ model of Ambal et al. [10], where intra-node concurrency follows the simpler SC model [43], while inter-node concurrency is analogous to that of RDMA$^{\text{TSO}}$. We then identify two sets of sufficient constraints that, if satisfied, ensure the robustness of RDMA$^{\text{SC}}$ programs. Our proposed constraints are purely *syntactic*, in that they do not require an understanding of the complex RDMA semantics and can be established by simply checking the syntax of the program. The first

set of constraints is restrictive, but can be applied to any RDMA program. The second relaxes the requirements of the first, but requires the RDMA network to follow a tree topology. Our conditions enable a number of useful paradigms for RDMA programs such as the server-client model, which we show can be used for automatically translating existing concurrent algorithms to distributed ones over RDMA, as well as for modelling star network topologies used e.g. in Local Area Networks (LAN). Finally, we adapt our results to the RDMA$^{\text{TSO}}$ model and accordingly propose analogous syntactic and topological constraints.

Outline. In §2 we present an intuitive account of the weak RDMA semantics through examples and discuss how we ensure robustness through syntactic constraints. In §3 we present our formal RDMA$^{\text{SC}}$ model. In §4 we establish sufficient syntactic conditions that ensure the robustness of RDMA$^{\text{SC}}$ programs. In §5 we apply these findings to tree-shaped network topologies, offering a further streamlined set of conditions under RDMA$^{\text{SC}}$. We discuss related work in §6. The proofs of all theorems stated in this paper, as well as the extension of all our results to the RDMA$^{\text{TSO}}$ model, are available in the extended version [11].

2 Overview

We present an intuitive account of RDMA semantics through several examples, showing the counter-intuitive and unexpected behaviours they can exhibit due to possible *instruction reorderings* (§2.1). We then discuss how we can tame this complexity by introducing *syntactic constraints* that, if fulfilled, prohibit problematic instruction reorderings, pre-empting unexpected behaviours and thus simplifying the task of reasoning about RDMA programs for developers (§2.2).

2.1 RDMA Semantics at a Glance

Consistency (Concurrency) Models and *Weak* Behaviours. In the literature of shared-memory concurrent (multi-threaded) programming, the set of possible behaviours (i.e. semantics) of a concurrent program is defined via a *consistency model* (a.k.a. memory model or concurrency model), with a number of such models available in different domains such as hardware architectures (e.g. Intel and ARM) and programming languages (e.g. C/C++ and Java). The most well-known and intuitive consistency model is *sequential consistency* (SC, a.k.a. interleaving concurrency) [43], where the instructions are interleaved in program order. That is, under SC the instructions in each thread cannot be *reordered*. While simple, SC is *too strong* in that it precludes many common hardware/compiler optimisations and thus unduly hinders performance. As such, modern hardware architectures and programming languages adhere to *weaker*, more lenient models, admitting more behaviours than SC. In this context a program behaviour (outcome) is referred to as *weak*, if it is not allowed under SC. Such weak behaviours can typically be understood in terms of instruction reorderings within a thread or visibility delays (where the effects of an instruction (e.g. a write) is not observed at the same time by all threads), both of which are disallowed under SC.

Conceptual RDMA Model. We model concurrent RDMA programs running over a network of *nodes* (i.e. computers), where each node hosts zero, one, or more threads, and each thread can *directly* access remote memory of other nodes through its *network interface card* (NIC). As we discuss below, RDMA programs exhibit three sources of weak behaviours: 1. *CPU* weak behaviours, due to the usual interactions (and reordering) of multiple threads on a single node; 2. *intra-thread* weak behaviours, due to RDMA operations being reordered or delayed; and 3. *inter-node* weak behaviours, due to multiple nodes executing concurrently. Here we focus on the latter two sources as they are specific to RDMA programs, and discuss how such weak behaviours may be prevented.

CPU Concurrency. RDMA enacts data transfers between nodes via the NIC subsystems of the constituent nodes, which are independent from the CPU subsystems. Consequently, the RDMA technology can be combined with different CPU architectures governed by different memory models (e.g. TSO or ARM). The first validated formal model of RDMA programs, RDMA^{TSO} [10], assumes that CPU concurrency is governed by the TSO model [68]. To simplify our presentation and not distract the reader from the RDMA complexities by the *orthogonal* intricacies of CPU concurrency, we present the simpler RDMA^{SC} model, where CPU concurrency follow the stronger SC model [43]. We generalise our results to RDMA^{TSO} in the extended version [11].

Almost all weak behaviours introduced by RDMA stem from the NIC and are independent of CPU concurrency (i.e. CPU and RDMA concurrency can often be decoupled). As such, the distinction between RDMA^{TSO} and RDMA^{SC} is often irrelevant, in which case we write RDMA^{*} to encompass both models. In particular, in this overview section we focus on nodes with *at most one thread each*, i.e. with no CPU concurrency, so all behaviours discussed below hold of both RDMA^{SC} and RDMA^{TSO} (i.e. for RDMA^{*}). Note that this is *merely a presentational choice* we have made in this section, and our formal models, theorems, and examples in subsequent sections also account for CPU concurrency.

Litmus Test Outcome Notation. We frequently present small representative examples (known as litmus tests in the literature). In each example, the outcomes annotated with ✓ are allowed by the RDMA model under discussion, while those annotated with ✗ are disallowed.

Remote Direct Memory Access (RDMA). RDMA programs comprise operations that access remote memory, as well as various synchronisation operations. As such, programming RDMA networks is conceptually similar to shared memory systems. To distinguish remote (RDMA) operations from CPU ones, we refer to RDMA reads and writes as *get* and *put* operations, respectively. To distinguish local and remote memory locations, we assume nodes do not reuse location names, we write x^n for a location on a remote node n, and write x for a location on the local node. A put operation is of the form $x^n := y$ and consists of reading from a local location y and writing to a remote location x on n. Similarly, a get operation is of the form $x := y^n$ and consists of reading from a remote location y on n and writing to a local location x. We write $\overline{n}$ to identify

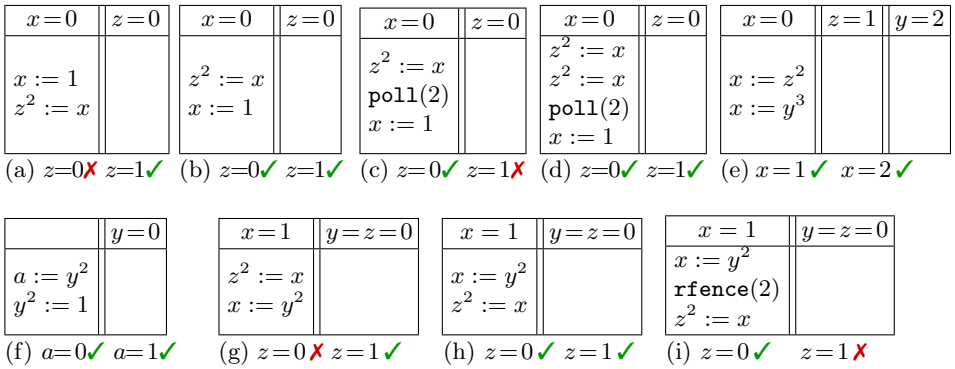


Fig. 1: Sequential RDMA* litmus tests, where each column (separated by ||) denotes a distinct node, the statement on the top line of each column denotes the initial values of locations.

a node other than n. When node n issues a remote operation to be executed on node $\overline{n}$, we state that the operation is *by* n *towards* $\overline{n}$.

Sequential (Single-Threaded) RDMA* Behaviours. When a thread issues a get or put operation, it is handled by the NIC, in contrast to local reads and writes handled by the CPU. As such, the interaction between CPU and remote operations lead to further behaviours even within a *sequential* (single-threaded) program. We demonstrate this in the examples of Fig. 1, where each column represents a distinct node, numbered from left to right starting from 1. For instance, the example in Fig. 1a comprises a single thread on node 1 (the left-most column) that writes to the local location x ($x := 1$) and puts x towards the remote location z on node 2 ($z^2 := x$).

Intuitively, when a thread t on n issues remote operations towards node $\overline{n}$, one can view these remote operations as if being executed by a thread running *in parallel* to t. As such, when a remote operation *follows* a CPU one, the order of the two operations is preserved since the parallel thread is spawned only after the CPU operation is executed. This is illustrated in Fig. 1a. By contrast, when a remote operation *precedes* a CPU one, the remote operation is performed by a 'separate thread' run in parallel to the later CPU operation in the main thread, and thus may execute before or after the CPU operation, meaning that in the latter case the execution order is not preserved. This is illustrated in Fig. 1b.

Therefore, before using the result of a get or reusing the memory location of a put, it is desirable to avoid such reorderings and to wait for the remote operation to complete. This can be done through a CPU *poll* operation, `poll`(n), that blocks until the *earliest* (in program order) remote operation towards node n has completed. This is shown in Fig. 1c, obtained from Fig. 1b by inserting a poll after the remote operation: `poll`(2) waits for $z^2 := x$ to complete before proceeding with $x := 1$, and thus $z^2 := x$ can no longer be reordered after $x := 1$.

Note that each `poll`(n) waits for *only one* (the earliest) and *not all* pending remote operations towards n to complete. For instance, in Fig. 1d, `poll`(2) only blocks until the *first* $z^2 := x$ is complete, and thus $z = 1$ is once again possible.

$y{=}0$	$x{=}0$
$x^2 := 1$ $a := y$	$y^1 := 1$ $b := x$

(a) $a{=}b{=}0$ ✓

$x{=}0$	$y{=}0$
$a := y^2$ $x := 1$	$b := x^1$ $y := 1$

(b) $a{=}b{=}1$ ✓

$x{=}0$	$y{=}0$
$a := y^2$ poll(2) $x := 1$	$b := x^1$ poll(1) $y := 1$

(c) $a{=}b{=}1$ ✗

$y{=}0$	$x{=}0$
$x^2 := 1$ poll(2) $a := y$	$y^1 := 1$ poll(1) $b := x$

(d) $a{=}b{=}0$ ✓

$y{=}w{=}0$	$x{=}z{=}0$
$x^2 := 1$ $c := z^2$ poll(2) poll(2) $a := y$	$y^1 := 1$ $d := w^1$ poll(1) poll(1) $b := x$

(e) $a{=}b{=}0$ ✗

Fig. 2: Concurrent RDMA* litmus tests.

Two remote operations towards *different* nodes are independent and can execute in either order, as illustrated in Fig. 1e. The only way to prevent this reordering is to poll the first operation before running the second.

The ordering guarantees on remote operations towards the *same* node are stronger and only certain reorderings are allowed. Recall that a put operation $x^n := y$ comprises two steps: a local read (on y) and a remote write (on x^n). Similarly, a get operation $x := y^n$ comprises two steps: a remote read (on y^n) and a local write (on x). Intuitively, NIC operations follow the *precedence* order: i) local read; ii) remote write; iii) remote read; iv) local write.

If a step with a higher precedence (e.g. a local read) is in program order before one with a lower precedence (e.g. a local write), then their order is preserved and they cannot be reordered. This is illustrated in Fig. 1g. Otherwise the order is not necessarily preserved and these steps can be reordered, as shown in Fig. 1h where an earlier local write on x can occur after the later local read.

As before, the reordering of the two remote operations in Fig. 1h can be prevented by polling the first operation before the second. However, polling is costly as it blocks the current thread, including the submission of remote operations towards any node. Alternatively, we can use a *remote fence*, `rfence`(n), that blocks *only* the NIC and *only* towards node n. This in turn ensures that earlier (before the fence) remote operations by the thread towards n are executed before later (after the fence) remote operations towards n. This is illustrated in Fig. 1i, obtained from Fig. 1h by inserting `rfence`(2) stopping the reordering.

Concurrent (Multi-Threaded) RDMA* Behaviours. The real power of RDMA comes from programs running on *different nodes*, introducing a wide range of weak behaviours. A network can comprise several nodes, each running several concurrent threads. We limit the examples of Fig. 2 to two nodes, each having a single thread.

As shown in Figs. 2a and 2b, well-known weak behaviours such as store buffering (Fig. 2a) and load buffering (Fig. 2b) are possible. This is because earlier RDMA operations can be delayed after later CPU operations.

As one could expect, most weak behaviours can be prevented by polling the remote operations as needed, as shown for load buffering in Fig. 2c. However, this strategy is not enough to prevent the store buffering weak behaviour, as show in Fig. 2d. This is because the specification of polling offers different guarantees for get and put operations. Polling a get operation $a := x^n$ offers the strong intuitive guarantee that the operation completed, i.e. the value of x^n is fetched

from node n and written to a. By contrast, polling a put operation $x^n := a$ does not guarantee the write on x^n has completed. When sending the value of a towards node n to be put in x^n, the remote NIC merely *acknowledges* having received the data, but this data may still reside in a *buffer* (i.e. the PCIe fabric) of the remote node, *pending* to be written x^n. Polling a put operation only awaits the acknowledgement of the data receipt. As such, it is possible to poll a put operation successfully before the associated remote write has fully completed. In the case of store buffering in Fig. 2d, it is possible for both poll operations to complete before the values of x and y are updated (to 1) in memory.

We also assume NICs are connected to memory though the *Peripheral Component Interconnect Express* (PCIe) fabric, the *de facto* standard for this category of hardware [10]. This ensures that (PCIe) reads cannot overtake (PCIe) writes. As such, a remote read *flushes* (commits) all pending remote writes to memory, and similarly on local memory. This can be used to prevent weak behaviours such as store buffering, as shown in Fig. 2e, obtained from Fig. 2d by adding additional gets and subsequently polling them. Polling a (seemingly unrelated) later get (e.g. $c := z^2$) ensures previous remote writes (e.g. $x^2 := 1$) have been committed to the remote memory.

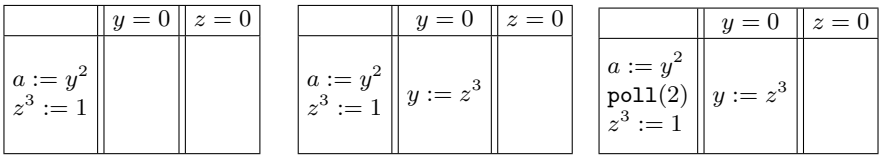
2.2 Robustness: Taming Weak RDMA* Behaviours

Given the permissive nature of the RDMA* semantics and the numerous weak behaviours it exhibits (even in the case of single-threaded programs), the task of writing *correct* RDMA programs is laborious. Reasoning about concurrent programs is already challenging even in the absence of weak behaviours. Accounting for potential instruction reorderings (which requires experience with RDMA* semantics) introduces yet another layer of complexity for developers.

As such, we should ideally enable reasoning about RDMA programs under a simpler, more intuitive model such as SC (sequential consistency [43]). Specifically, to simplify program reasoning, a common approach is to ensure *robustness*. A program P is *robust* under a consistency model CM, if its set of possible behaviours under CM coincide with those of its behaviours under SC; i.e. P is robust if it exhibits no weak behaviours. If a program is robust, then we can reason about it *as if* it were executed under SC, without considering the complexities of RDMA*.

To ensure robustness, we must prevent *observable* reorderings, i.e. those leading to weak behaviours. We can achieve this through *syntactic* requirements (e.g. by inserting sufficient remote fences and poll operations). A naive solution is to wait for each remote operation to fully complete before proceeding further, thereby preventing all reorderings. Unfortunately, this *serialises* these operations, and thus defeats the benefits of RDMA, which is designed to parallelise CPU instructions and data transfers by offloading them to the NIC. Instead, we should account for the RDMA* semantics and only add restrictions when necessary, while allowing non-observable reorderings.

Certain reorderings are observable even when considering a single thread in isolation, as in the examples of Figs. 1b, 1e, 1f, and 1h. Specifically, these exam-



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

Fig. 3: Examples showing that necessary restrictions depend on other threads.

ples contain *data races within a single thread*. Beyond robustness, these patterns should be avoided in any sensible program. However, most weak behaviours arise from the interaction of several threads. For instance, in the single-threaded example of Fig. 3a, although the two remote operations $a := y^2$ and $z^3 := 1$ on node 1 may be reordered, this reordering is *not observable*: it does not lead to additional weak behaviours, and thus no additional constraints are necessary for robustness. By contrast, in the multi-threaded variant of Fig. 3b (with a thread on node 2), nodes 2 and 3 can exchange data and thus we can observe the weak behaviour $a = 1$ due to this reordering. As such, to prohibit this, we must prevent the two operations on node 1 from being reordered, e.g. by polling the first operation, as shown in Fig. 3c.

As seen before, preventing reorderings can be done in different ways. In cases like Fig. 1i, a remote fence is enough. In cases like Fig. 2e, we need dummy get operations. Determining when and how to prevent reorderings is not straightforward. As illustrated in the examples of Fig. 3, it cannot be done *thread-locally*: one must account for the communication between other nodes and thus must take the whole program into account. This raises two questions:

- How do we prevent weak behaviours through simple *purely syntactic* restrictions? Specifically, how can we ensure that a program has enough constraints (e.g. polls) to *prevent weak behaviours*, and how do we make sure that waiting for a specific remote operation (as in Fig. 3a) is unnecessary?
- How do we structure RDMA programs to *minimise* the amount of necessary restrictions in order to maintain efficient implementations?

We set out to answer these questions in the remainder of this paper. Specifically, after defining several formal preliminaries in §3, we present a theorem in §4 stating *sufficient syntactic conditions* guaranteeing robustness (i.e. the absence of weak behaviours). In §5 we then build on this theorem and present a useful RDMA network topology where fewer limitations are necessary to prevent weak behaviours. Notably, following our prescribed network topology ensures that it is *never necessary to poll* a remote operation to prevent multi-threaded weak behaviours.

3 RDMA^{SC}: A Declarative Semantics for RDMA Programs

We present the syntax of RDMA programs (taken from [10]) in §3.1. In §3.2 we then present a formal declarative semantics for our RDMA^{SC} model. As we

describe in the extended version [11], we obtain RDMA$^{\text{SC}}$ by *strengthening* the RDMA$^{\text{TSO}}$ model of Ambal et al. [10] whereby we make a few simple adjustments to ensure that local (CPU) concurrency follows the SC rather than TSO model.

3.1 RDMA$^{\text{SC}}$: Programming Language

Nodes and Threads. We consider a system with N nodes and M threads in total across all nodes. Let $\mathsf{Node} = \{1, \ldots, N\}$ and $\mathsf{Tid} = \{1, \ldots, M\}$ denote the sets of *node* and *thread identifiers*, respectively. We use n and t to range over Node and Tid, respectively. Given a node n, we write $\overline{n}$ to range over $\mathsf{Node} \setminus \{n\}$. Each thread $t \in \mathsf{Tid}$ is associated with a node, written $n(t)$.

Memory Locations. Each node n has a set of *locations*, Loc_n, accessible by all nodes. We define $\mathsf{Loc} \triangleq \biguplus_n \mathsf{Loc}_n$ and $\mathsf{Loc}_{\overline{n}} \triangleq \mathsf{Loc} \setminus \mathsf{Loc}_n$. We use x^n, y^n, z^n, w^n and $x^{\overline{n}}, y^{\overline{n}}, z^{\overline{n}}, w^{\overline{n}}$ to range over Loc_n and $\mathsf{Loc}_{\overline{n}}$, respectively. When the choice of n is clear, we write x for x^n and $\overline{x}$ for $x^{\overline{n}}$. For clarity, we use distinct location names across nodes and write $n(x)$ for the unique $n \in \mathsf{Node}$ where $x \in \mathsf{Loc}_n$. We assume all locations can be accessed by all threads on all nodes. However, for readability, we use a, b, c, and d for (private) locations that are only accessed by a single thread (on a single node).

Values and Expressions. We assume a set of values, Val, with $\mathbb{N} \subseteq \mathsf{Val}$, and use v to range over Val. We assume a language of expressions over Val and Loc, and elide its exact syntax and semantics. We use e to range over expressions, and e^n to range over expressions whose locations are all included in Loc_n.

Sequential Commands and Programs. *Sequential* programs on node n are described by the C^n grammar below and include primitive commands (c^n), sequential composition ($\mathsf{C}_1^n; \mathsf{C}_2^n$), non-deterministic choice ($\mathsf{C}_1^n + \mathsf{C}_2^n$, executing either C_1^n or C_2^n), and non-deterministic loops (C^{n*}, executing C^n any number of times). A (concurrent) *program*, P, is a map from thread identifiers to commands, associating each thread $t \in \mathsf{Tid}$ with a command on node $n(t)$.

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

Primitive commands include *CPU* (cc^n) and *RDMA* (rc^n) operations. A CPU operation on n may be a no-op ($\mathtt{skip}$), an assignment to a local location ($x := e$), an assumption on the value of a local location ($\mathtt{assume}(x = v)$ and $\mathtt{assume}(x \neq v)$), an atomic CAS ('compare-and-set') operation ($x := \mathtt{CAS}(y, e_1, e_2)$), or a 'poll', $\mathtt{poll}(\overline{n})$, that awaits the completion notification of the earliest put/get that is pending (not yet acknowledged). An RDMA operation may be (i) a 'get', $x := \overline{y}$, reading from remote location $\overline{y}$ and writing the result to local location x; (ii) a 'put', $\overline{y} := x$, reading from local location x and writing the result to remote location $\overline{y}$; or (iii) a 'remote fence', $\mathtt{rfence}(\overline{n})$, which ensures that all

later (in program order) RDMA operations towards $\overline{n}$ will await the completion of all earlier RDMA operations towards $\overline{n}$. $\texttt{poll}(\overline{n})$ is executed by the CPU and blocks its thread (and prevents the requests of later remote operations), while $\texttt{rfence}(\overline{n})$ blocks the NIC for the execution of remote operations towards $\overline{n}$.

3.2 RDMA$^{\text{SC}}$: Declarative Semantics

Events and Executions. In the literature of declarative models, the traces of a program are commonly represented as a set of *executions*, where an execution is a graph comprising: i) a set of *events* (graph nodes); and ii) a number of relations on events (graph edges). Each event is associated with the execution of a primitive command (in PComm) and is a tuple (ι, t, l), where ι is the (unique) *event identifier*, $t \in \mathsf{Tid}$ identifies the executing thread, and $l \in \mathsf{ELab}$ is the *event label*, defined below.

Definition 1 (Labels and events). *An* event, $\mathsf{e} \in \mathsf{Event}$, *is a triple* (ι, t, l), *where* $\iota \in \mathbb{N}$, $t \in \mathsf{Tid}$ *and* $l \in \mathsf{ELab}_{n(t)}$. *The set of* event labels *is* $\mathsf{ELab} \triangleq \bigcup_n \mathsf{ELab}_n$ *for all nodes* n. *An* event label *of* n, $l \in \mathsf{ELab}_n$, *is a tuple of one of the following forms:*

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

Each event label denotes whether the associated primitive command is handled by the NIC (left column, prefixed with $\texttt{n}$), or the CPU (right column). A poll instruction is handled by the CPU. A put operation $x^{\overline{n}} := y^n$ by node n towards node $\overline{n}$ comprises a NIC local read from y^n and a NIC remote write on $x^{\overline{n}}$ and is thus modelled as two events with labels $\texttt{nlR}(y^n, v, \overline{n})$ and $\texttt{nrW}(x^{\overline{n}}, v)$, where v denotes the value read from y^n and written to $x^{\overline{n}}$. Similarly, a get $x^n := y^{\overline{n}}$ is modelled as two events with labels of the form $\texttt{nrR}(y^{\overline{n}}, v)$ and $\texttt{nlW}(x^n, v, \overline{n})$.

CPU operations are modelled by events as expected. A successful operation $x := \texttt{CAS}(y, v_1, v_2)$ is modelled by two events with labels $\texttt{CAS}(y, v_1, v_2)$ and $\texttt{lW}(x, v_1)$. An unsuccessful $x := \texttt{CAS}(y, v_1, v_2)$ operation is modelled by a CPU read instead: $\texttt{lR}(y, v)$ and $\texttt{lW}(x, v)$, with $v \neq v_1$.

We write $\mathsf{type}(l)$, $\texttt{loc}(l)$, $v_{\text{r}}(l)$, $v_{\text{w}}(l)$, and $\overline{n}(l)$ for the type (e.g. $\texttt{lR}$), location, read value, write value, and remote node of l, where applicable; e.g. $\texttt{loc}(\texttt{nlR}(x^n, v_{\text{r}}, \overline{n})) = x^n$ and $\overline{n}(\texttt{nlR}(x^n, v_{\text{r}}, \overline{n})) = \overline{n}$. We lift these functions to events as expected. We write $\iota(\mathsf{e})$, $t(\mathsf{e})$, $l(\mathsf{e})$ to project the corresponding components of an event $\mathsf{e} = (\iota, t, l)$, and write $n(\mathsf{e})$ for the node $n(t(\mathsf{e}))$ of an event.

Queue Pairs. As mentioned in §2 (see Fig. 1e), two remote operations by the same thread towards different remote nodes can be reordered. When using RDMA, each thread establishes a communication channel, called a *queue pair*, towards each remote node. The intuition is that operations on different queue pairs are independent and can always be reordered. Different threads, even on the same node, create different queue pairs to connect to the same remote node.

Notation. Given a relation r and a set A, we write r^+ for the transitive closure of r; r^{-1} for the inverse of r; $r|_A$ for $r \cap (A \times A)$; and $[A]$ for the identity relation on A, i.e. $\{(a,a) \mid a \in A\}$. We write $r_1; r_2$ for their relational composition: $\{(a,b) \mid \exists c.\, (a,c) \in r_1 \wedge (c,b) \in r_2\}$. When r is a strict partial order, we write $r|_{\text{imm}}$ for the *immediate* edges in r, i.e. $r \setminus (r;r)$. Given a set of events E and a location x, we write E_x for $\{\mathsf{e} \in E \mid \mathtt{loc}(\mathsf{e}) = x\}$. Given a set of events E and a label type $\mathtt{X}$, we write $E.\mathtt{X}$ for $\{\mathsf{e} \in E \mid \mathsf{type}(\mathsf{e}) = \mathtt{X}\}$, and define its sets of *reads* as $E.\mathcal{R} \triangleq E.\mathtt{lR} \cup E.\mathtt{CAS} \cup E.\mathtt{nlR} \cup E.\mathtt{nrR}$, *writes* as $E.\mathcal{W} \triangleq E.\mathtt{lW} \cup E.\mathtt{CAS} \cup E.\mathtt{nlW} \cup E.\mathtt{nrW}$, *CPU events* as $E^{\text{cpu}} \triangleq E.\mathtt{lW} \cup E.\mathtt{lR} \cup E.\mathtt{CAS} \cup E.\mathtt{P}$, and *NIC writes* as $E.\mathtt{nW} \triangleq E.\mathtt{nlW} \cup E.\mathtt{nrW}$. We define the '*same-location*' relation as $\mathsf{sloc} \triangleq \{(\mathsf{e},\mathsf{e}') \in \mathsf{Event}^2 \mid \mathtt{loc}(\mathsf{e}) = \mathtt{loc}(\mathsf{e}')\}$; the '*same-thread*' relation as $\mathsf{sthd} \triangleq \{(\mathsf{e},\mathsf{e}') \in \mathsf{Event}^2 \mid t(\mathsf{e}) = t(\mathsf{e}')\}$; and the '*same-queue-pair*' relation as $\mathtt{sqp} \triangleq \{(\mathsf{e},\mathsf{e}') \in \mathsf{Event}^2 \mid t(\mathsf{e}) = t(\mathsf{e}') \wedge \overline{n}(\mathsf{e}) = \overline{n}(\mathsf{e}')\}$. We use $\mathtt{sqp}$ for events on the same queue pair, i.e. by the same thread and towards the same remote node. Note that $\mathtt{sqp} \subseteq \mathsf{sthd}$ and that sloc, sthd, and $\mathtt{sqp}$ are all symmetric. For a set of events E, we write $E.\mathsf{sloc}$ for $\mathsf{sloc}|_E$; similarly for $E.\mathsf{sthd}$ and $E.\mathtt{sqp}$.

Definition 2 (Pre-executions). *A tuple $\mathcal{G} = \langle E, \mathsf{po}, \mathsf{pf} \rangle$ is a* pre-execution *of a program if:*

- $E \subseteq \mathsf{Event}$ *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 $E^0 \times (E \setminus E^0) \subseteq \mathsf{po}$, and such that:*
  - *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 $w \in G.\mathtt{nrW}$ (resp. $w \in G.\mathtt{nlW}$), then $(r,w) \in \mathsf{po}|_{imm}$ for some $r \in G.\mathtt{nlR}$ ($r \in G.\mathtt{nrR}$).*
  - *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 $v_r(r) = v_w(w)$.*
- $\mathsf{pf} \subseteq E.\mathtt{nW} \times E.\mathtt{P}$ *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). Also, Poll events poll-from the oldest non-polled remote operation on the same queue pair:*
  if $w_1 \in G.\mathtt{nW}$ and $w_1 \xrightarrow{\mathsf{po} \cap \mathtt{sqp}} w_2 \xrightarrow{\mathsf{pf}} p_2$, then there exists p_1 such that $w_1 \xrightarrow{\mathsf{pf}} p_1 \xrightarrow{\mathsf{po}} p_2$.

Pre-executions are constructed syntactically by induction on the structure of the corresponding program. This definition is standard and omitted.

Intuitively, a pre-execution can also be seen as a trace of the execution: for each thread t, po restricted to t is a total order, and so $\langle E, \mathsf{po} \rangle$ is fundamentally

a sequence of events for each thread. In this view, pf should be considered a well-formedness condition: each prefix of the trace needs to have at least as many remote operations as poll operations. So $\langle E, \mathsf{po}, \mathsf{pf}\rangle$ can be seen as providing a well-formed trace for each thread. We later define robustness conditions on pre-executions, and as such they can also be considered conditions on traces.

We next extend the notion of a pre-execution to an *execution* by choosing explicitly how the different events interact.

Definition 3 (Executions). $G = \langle E, \mathsf{po}, \mathsf{pf}, \mathsf{rf}, \mathsf{mo}, \mathsf{nfo}\rangle$ *is an* execution *if:*

- $\langle E, \mathsf{po}, \mathsf{pf}\rangle$ *is a pre-execution.*
- $\mathsf{rf} \subseteq E.\mathcal{W} \times E.\mathcal{R}$ *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* $E_x^0 \times (E.\mathcal{W}_x \setminus E_x^0) \subseteq \mathsf{mo}_x$ *describing the order in which writes on* x *reach the memory.*
- $\mathsf{nfo} \subseteq E.\mathtt{sqp}$ *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* $a \in E.\mathtt{nrR}, b \in E.\mathtt{nrW}$, *then* $(a,b) \in \mathsf{nfo} \cup \mathsf{nfo}^{-1}$.

We define the *reads-before* relation as $\mathsf{rb} \triangleq (\mathsf{rf}^{-1}; \mathsf{mo}) \setminus [E]$, relating each read r to writes that are mo-after the write r reads from. Given a (pre-)execution G (resp. $\mathcal{G}$), we use the '$G.$' prefix to project its various components (e.g. $G.\mathsf{rf}$) and derived relations (e.g. $G.\mathsf{rb}$). When the context is clear, we drop the prefix.

PCIe guarantees that a NIC local read (`nlR`) propagates all pending NIC local writes (`nlW`) (processed by the same queue pair) to memory, while a NIC remote read (`nrR`) propagates all pending NIC remote writes (`nrW`) (processed by the same queue pair) to memory. We model this total order through the nfo relation, stipulating that all NIC local reads and writes (resp. all NIC remote reads and writes) on the same queue pair be totally ordered.

Issue and Observation Points. In what follows we distinguish between when an instruction is *issued* and when it is *observed*. Intuitively, an instruction is issued when it is processed by the CPU or the NIC, and it is observed when its effect is propagated to memory. As such, since NIC writes can be delayed and have an observable effect on memory, the time points at which they are issued and observed may differ. Since we assume CPUs follow the strong SC memory model, CPU writes are issued and observed at the same time. However, the local (resp. remote) write of a get (resp. put) is issued when it is processed by the NIC and sent to the PCIe fabric, and observed when it is propagated to memory. All other events are *instantaneous* in that *either* they do not have an observable effect on memory (e.g. reads), *or* their effect is written to memory *immediately* (e.g. CAS operations and CPU writes). Given a set of events E, we thus define the set of *instantaneous events in* E as $E.\mathtt{Inst} \triangleq E \setminus (E.\mathtt{nlW} \cup E.\mathtt{nrW})$. Intuitively, the effects of NIC local writes and NIC remote writes (labelled `nlW` and `nrW`) can be delayed in the PCIe fabric and are thus excluded from the set

Later in Program Order

Earlier in PO	ippo	1	2	3	4	5	6
		E^{cpu}	nlR	nrW	nrR	nlW	nF
A	E^{cpu}	✓	✓	✓	✓	✓	✓
B	nlR	✗	sqp	sqp	sqp	sqp	sqp
C	nrW	✗	✗	sqp	sqp	sqp	sqp
D	nrR	✗	✗	✗	✗	sqp	sqp
E	nlW	✗	✗	✗	✗	sqp	sqp
F	nF	✗	sqp	sqp	sqp	sqp	sqp

Later in Program Order

Earlier in PO	oppo	1	2	3	4	5	6
		E^{cpu}	nlR	nrW	nrR	nlW	nF
A	E^{cpu}	✓	✓	✓	✓	✓	✓
B	nlR	✗	sqp	sqp	sqp	sqp	sqp
C	nrW	✗	✗	sqp	sqp	sqp	✗
D	nrR	✗	✗	✗	✗	sqp	sqp
E	nlW	✗	✗	✗	✗	sqp	✗
F	nF	✗	sqp	sqp	sqp	sqp	sqp

Fig. 4: The RDMA$^{\text{SC}}$ ordering constraints on ippo (left) and oppo (right), 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.

of instantaneous events. Note that the observation point either follows the issue point (for NIC writes), or coincides (for instantaneous events).

We next define the '*issue-preserved program order*', ippo, as the subset of po edges (ippo $\subseteq$ po) that must be preserved when issuing instructions. That is, if two events are ippo-related, then they must be issued in program order; otherwise they may be processed in either order. The left table of Fig. 4 describes which po edges are included in ippo, where ✓ denotes that the two instructions are ippo-related (i.e. they must be issued in program order), ✗ denotes that they are not ippo-related (i.e. they may be issued out of order) and `sqp` denotes that they are ippo-related iff they are on the same queue pair. For instance, when a CPU instruction is followed by anything, they are issued in order (line A); but when a NIC instruction is followed by a CPU one, they may be reordered (cells B1-F1).

Analogously, we define the '*observation-preserved program order*', oppo, as the subset of po edges (oppo $\subseteq$ po) that must be preserved when observing the effects of instructions. I.e., if two events are oppo-related, then they become observable in program order in RDMA$^{\text{SC}}$; otherwise they may become observable in either order. The right table of Fig. 4 describes which po edges are included in oppo. The two tables differ in cells C6 and E6. This is because NIC writes can be delayed, and remote fences do not guarantee propagation to memory.

RDMA$^{\text{SC}}$ Consistency. The notion of executions (Def. 3) imposes very few constraints on the po, pf, rf, mo, and nfo relations. Such restrictions and thus the permitted behaviours of a program are determined by defining the set of *consistent* executions, defined below.

Definition 4 (RDMA$^{\text{SC}}$-consistency). *An execution* $\langle E, \mathsf{po}, \mathsf{pf}, \mathsf{rf}, \mathsf{mo}, \mathsf{nfo}\rangle$ *is* RDMA$^{\text{SC}}$-consistent *iff* ib *and* ob *are irreflexive, where:*

$$\mathsf{ib} \triangleq \big(\mathsf{ippo} \cup \mathsf{rf} \cup \mathsf{pf} \cup \mathsf{nfo}\big)^{+} \qquad \text{('issued-before')}$$

$$\mathsf{ob} \triangleq \big(\mathsf{oppo} \cup \mathsf{rf} \cup ([\texttt{nlW}]; \mathsf{pf}) \cup \mathsf{nfo} \cup \mathsf{rb} \cup \mathsf{mo} \cup ([\texttt{Inst}]; \mathsf{ib})\big)^{+} \qquad \text{('observed-before')}$$

The ib (resp. ob) relation is an extension of ippo (resp. oppo), describing the issue (resp. observation) order across the instructions of different threads and nodes. RDMA^{SC}-consistency requires that ib and ob be irreflexive (i.e. yield strict partial orders as they are defined transitively).

The rf (resp. pf) component in ib states that if e reads from (resp. polls from) w, then w must have been issued before e. Recall that nfo totally orders the nlR/nlW and nrR/nrW operations on the same queue pair and is thus in ib. The rf component in ob states that if a read r reads from a write w, then the write has reached memory. This is because reads can only read the main memory and not auxiliary buffers. The [nlW]; pf component states that if p polls from a NIC local write w, then w must have left the PCIe fabric and reached the memory. Note that this is not the case for nrW events: polling an nrW event w might succeeds when w is still in the remote PCIe fabric before reaching the remote memory. The nfo in ob can be justified as in the case of ib. The rb component in ob ensures that a read r on x observes the latest write on x that has reached the memory. As mo describes the order in which the writes on each location reach the memory, it is included in ob. Let (τ_i, τ_o) be the issue and observation points of e and (τ_i', τ_o') be those of e′. The [Inst]; ib in ob ensures that if $\mathsf{e} \xrightarrow{\mathsf{ib}} \mathsf{e}'$ (i.e. $\tau_i < \tau_i'$) and e is instantaneous ($\tau_i = \tau_o$), then $\tau_o = \tau_i < \tau_i' \leq \tau_o'$, i.e. $\mathsf{e} \xrightarrow{\mathsf{ob}} \mathsf{e}'$.

4 Robustness of RDMA^{SC} Programs

In the traditional setting of CPU concurrency (where all threads execute CPU instructions), the most intuitive consistency model is *sequential consistency* (SC) [43]. While SC is too strong—in that disallowing *all* reorderings does not enable efficient implementations—it provides an intuitive and commonly understood model, making it easier for developers to reason about their programs.

Although none of the existing well-known consistency models follow SC by default, programmers typically address this difficulty by focusing on *robust* implementations of algorithms. Specifically, a program is robust under a weak consistency model CM if every possible behaviour of the program under CM is also an allowed behaviour under SC. In our model, this is defined as follows.

Definition 5 (SC-consistency and RDMA^{SC}-robustness). *Given an execution* $\langle E, \mathsf{po}, \mathsf{pf}, \mathsf{rf}, \mathsf{mo}, \mathsf{nfo} \rangle$*, its associated* sequential-consistency relation *is defined as* $\mathsf{sc} \triangleq (\mathsf{po} \cup \mathsf{rf} \cup \mathsf{rb} \cup \mathsf{mo})$*. An execution* G *is* SC-consistent *iff* $G.\mathsf{sc}$ *is acyclic. A pre-execution is* robust under RDMA^{SC} *iff all of its* RDMA^{SC}*-consistent executions (Def. 4) are also SC-consistent.*

Our aim here is to provide guidelines to ensure the robustness of RDMA^{SC} programs. That is, we identify a number of *syntactic* requirements such that if a program fulfils them, then the behaviours of the program under RDMA^{SC} coincide with its behaviours under SC; i.e. the program does not exhibit any weak behaviours brought about by observable reorderings.

There are two complementary approaches to achieve robustness. The first is to structure the program in a way that limits the very existence of problematic

Later in po

Earlier in po	gb		different queue pair					same queue pair			
			1	2	3	4	5	6	7	8	9
			CPU	nlR	nrW	nrR	nlW	nlR	nrW	nrR	nlW
A	CPU		✓	✓	✓	✓	✓	N/A			
B	nlR		P	P	P	P	P	✓	✓	✓	✓
C	nrW		GP	GP	GP	GP	GP	GP	✓	✓	✓
D	nrR		P	P	P	P	P	F	F	F	✓
E	nlW		P	P	P	P	P	F	F	P	✓

Fig. 5: Constraints necessary to guarantee that a pair of po-related events in $\mathcal{R} \cup \mathcal{W}$ will be ob-related for any consistent execution. CPU denotes local events in $\texttt{lW} \cup \texttt{lR} \cup \texttt{CAS}$. The ✓ denotes that no additional constraint is needed and that the events are already in ob. P denotes that the earlier operation must be polled before executing the later one. F denotes that either the earlier operation must be polled (similar to P) or that a remote fence must be inserted between the two operations. GP denotes that a get operation and its associated poll on the first queue pair must be inserted between the two operations.

cases. The second is to extend the program with enough restrictions (e.g. polls and remote fences) to prohibit reorderings. In the next section (§5) we focus on the former and provide a set of explicit guidelines to avoid most problematic cases by design. In this section we focus on the latter, and describe how to identify problematic cases and how to block them. In what follows, we present the general syntactic restrictions required to forbid the reordering opportunities for specific operations (§4.1). We then propose sufficient syntactic conditions that block observable reorderings, and we prove that these conditions imply robustness (§4.2). Finally, we discuss the limitations of this approach (§4.3).

4.1 A Syntactic Approach to Enforce the Program Order

One of our key results relies on enforcing the program order (i.e. blocking instruction reordering) in potentially observable cases. Recall that given an execution, the observed-before order (ob) describes when an event takes effect before another. That is, for $(\mathsf{e}_1, \mathsf{e}_2) \in \mathsf{po}$, when $\mathsf{e}_1 \xrightarrow{\mathsf{ob}} \mathsf{e}_2$ in an execution G, then they are not reordered in G. Our first aim here is to identify syntactic constraints that ensure that a specific pair of given instructions (of the same thread) are related by ob. However, in order to define syntactic constraints for robustness, we can only rely on the *syntax* of the program and not components such as rf or mo. Our syntactic constraints can only rely on the pre-execution components po and pf, and we cannot directly use the ob relation derived from a specific execution.

To this end, we first define the *guaranteed-before* relation, $\mathsf{gb} \subseteq \mathsf{po}$, describing when two instructions in the same thread are guaranteed to remain in order (and their reordering is blocked), as shown in Fig. 5. Specifically, if two instructions are related by oppo, then they are guaranteed to be observed in that order and

thus there is no need for additional restrictions; this is denoted by ✓ in cells A1–A5, B6–B9, C7–C9, D9, and E9 (*cf.* oppo in Fig. 4). For most other cases (noted P or F), polling the earlier instruction enforces the ordering. Recall that polling a NIC remote write does not guarantee its completion, and we need to add a 'dummy' get operation and its corresponding poll to ensure ordering (noted GP).

In most cases, when the two operations are on the *same* queue pair, then a remote fence is sufficient to enforce the ordering (noted F in D6–D8, E6–E7), and is a cheaper alternative to a poll. Perhaps surprisingly, a remote fence is not always sufficient: the two outliers are cells C6 and E8. For C6, consider the program $z^2 := x; \texttt{rfence}(2); w^2 := y$: the local value of y might be read before the value of z is changed. This is because `rfence` (2) (as with poll) only awaits the acknowledgement from the remote side which does not necessarily ensure that the first put has completed. For E8, consider $x := z^2; \texttt{rfence}(2); y := w^2$, where w^2 can be read before x is modified: `rfence` (2) only waits for the NIC local write ($x := v_z$) to be sent to the local PCIe fabric and thus the put operation ($y := w^2$) can start earlier than one could expect.

Definition 6 (guaranteed-before). *Given a pre-execution* $\mathcal{G} = \langle E, \mathsf{po}, \mathsf{pf}\rangle$, *its* guaranteed-before *order,* $\mathsf{gb} \subseteq \mathsf{po}$, *is defined as* $\mathsf{gb} \triangleq \mathsf{gb}_{\mathsf{base}}^{+}$, *with:*

$$
\begin{aligned}
\mathsf{gb}_{\mathsf{base}} \triangleq\ & \mathsf{oppo} && \text{(A1–A5,B6–B9,C7–C9,D9,E9 in Fig. 5)}\\
& \cup [\texttt{nlR}]; \mathsf{po}|_{imm}; [\texttt{nrW}]; \mathsf{pf} && \text{(B1–B5 in Fig. 5)}\\
& \cup [\texttt{nrW}]; (\mathsf{po} \cap \texttt{sqp}); [\texttt{nlW}]; \mathsf{pf} && \text{(C1–C6 in Fig. 5)}\\
& \cup [\texttt{nrR}]; \mathsf{po}|_{imm}; [\texttt{nlW}]; \mathsf{pf} && \text{(D1–D8 in Fig. 5)}\\
& \cup [\texttt{nrR}]; (\mathsf{po} \cap \texttt{sqp}); [\texttt{nF}]; (\mathsf{po} \cap \texttt{sqp}) && \text{(D6–D8 in Fig. 5)}\\
& \cup [\texttt{nlW}]; \mathsf{pf} && \text{(E1–E8 in Fig. 5)}\\
& \cup [\texttt{nlW}]; (\mathsf{po} \cap \texttt{sqp}); [\texttt{nF}]; (\mathsf{po} \cap \texttt{sqp}); [\texttt{nlR} \cup \texttt{nrW}] && \text{(E6–E7 in Fig. 5)}
\end{aligned}
$$

Given an execution $G = \langle E, \mathsf{po}, \mathsf{pf}, \mathsf{rf}, \mathsf{mo}, \mathsf{rb}\rangle$, we write $G.\mathsf{gb}$ for $\langle E, \mathsf{po}, \mathsf{pf}\rangle.\mathsf{gb}$. Finally, we prove that gb implies ob for any RDMA$^{\text{SC}}$-consistent execution (see the extended version [11] for the proof).

Theorem 1 (gb implies ob). *Given a pre-execution* $\langle E, \mathsf{po}, \mathsf{pf}\rangle$, *for all RDMA^{SC}-consistent executions* $G = \langle E, \mathsf{po}, \mathsf{pf}, \mathsf{rf}, \mathsf{mo}, \mathsf{nfo}\rangle$ *and all* $\mathsf{e}_1, \mathsf{e}_2 \in E$, *if* $(\mathsf{e}_1, \mathsf{e}_2) \in G.\mathsf{gb}$, *then* $(\mathsf{e}_1, \mathsf{e}_2) \in G.\mathsf{ob}$.

Given Theorem 1 above, we can use gb as a tool to enforce robustness. Specifically, whenever a program order pair $(\mathsf{e}_1, \mathsf{e}_2) \in \mathsf{po}$ may be reordered, we can add the prescribed fences to enforce $(\mathsf{e}_1, \mathsf{e}_2) \in \mathsf{gb}$ and thus block the reordering. The rest of this section describes *when* we should use this tool.

4.2 Conditions for Robustness under RDMA$^{\text{SC}}$

As mentioned before, blocking all instruction reorderings, i.e. by requiring $\mathsf{po} = \mathsf{gb}$, would enforce sequential consistency and thus robustness. However, this is too strict and highly impractical. Instead, we should ideally enforce gb selectively when needed and only prevent observable reorderings.

Two sources of weak behaviours. As presented in §2, RDMA^{SC} programs have two distinct sources of weak behaviours. These come from two different kinds of pairs of events (of the same thread): (1) pairs forming a data race on a certain location, e.g. $a := y^2; y^2 := 1$, as presented in Fig. 1f (copied below-left) and Figs. 1b, 1e, and 1h ; and (2) pairs whose reordering can be observed by other threads, e.g. $a := y^2; z^3 := 1$, as in the examples of Fig. 3b (copied below-right).

	$y=0$
$a := y^2$ $y^2 := 1$	

$a = 1$ ✓

	$y = 0$	$z = 0$
$a := y^2$ $z^3 := 1$	$y := z^3$	

$a = 1$ ✓

As such, stopping these two sources of weak behaviours would be enough to ensure robustness. Data races within a thread are *always* problematic, no matter the context, and we always need to block the reordering of such pairs (i.e. enforce gb to ensure the pair is ob-ordered in any execution). Pairs of the second kind cannot create weak behaviours by themselves, but they might allow weak behaviours depending on the rest of the program of other threads. In the next section (§5), we show conditions making sure that such pairs can never create weak behaviours by design. In this section, we focus on deciding whether such a pair might lead to a weak behaviour and, if so, how to block the reordering.

To formulate this intuition, we write $\mathsf{public}(x)$ to denote that x is a *public* location accessed by multiple threads, and given a set of events E, we define the set of public events in E as $E^{\mathsf{pub}} \triangleq \{\mathsf{e} \in E \mid \mathsf{public}(\mathtt{loc}(\mathsf{e}))\}$. We further define $E \setminus t \triangleq \{\mathsf{e} \in E \mid t(\mathsf{e}) \neq t\}$ for the set of events in E that are not by thread t. We can then formulate the two categories of weak behaviours above as two kinds of sc cycles: sc cycles on a single thread (1) and sc cycles on public events across threads (2), as formulated below (see the extended version for the full proof).

Theorem 2 (sc cycle decomposition). *Given a RDMA^{SC}-consistent execution $G = \langle E, \mathsf{po}, \mathsf{pf}, \mathsf{rf}, \mathsf{mo}, \mathsf{nfo}\rangle$, if $\exists \mathsf{e} \in E.\ \mathsf{e} \xrightarrow{G.\mathsf{sc}}{}^{+} \mathsf{e}$ (i.e. a cycle in G.sc), then:*

- *either there is a G.sc cycle on a single thread, i.e. $\exists \mathsf{e} \in E.\ \mathsf{e} \xrightarrow{G.\mathsf{sc} \cap \mathsf{sthd}}{}^{+} \mathsf{e}$;*
- *or there exists $\mathsf{e}_1, \mathsf{e}_2 \in E^{\mathsf{pub}}$ such that $\mathsf{e}_1 \xrightarrow{\mathsf{po} \setminus G.\mathsf{ob}} \mathsf{e}_2 \xrightarrow{(G.\mathsf{sc};[E^{\mathsf{pub}} \setminus t(\mathsf{e}_1)])^{+};G.\mathsf{sc}} \mathsf{e}_1$. That is, there is an sc cycle on public events, with two po-related events on some thread $t(\mathsf{e}_1)$ not related in ob, and where the rest of the cycle does not go through the events of $t(\mathsf{e}_1)$.*

The two kinds of problematic reorderings are tackled separately below, and Theorem 5 confirms the two resulting conditions are sufficient for robustness.

Preventing sc cycles from data races. As shown above, when an allowed reordering is part of a data race, it becomes observable independently from the context. Thus, we should always preclude this kind of reordering. Specifically, in Def. 7 below we present a *local data-race freedom* property to block data races within each thread and prevent single-threaded weak behaviours.

Definition 7 (Local DRF). *Given a pre-execution* $\langle E, \mathsf{po}, \mathsf{pf}\rangle$*, two events* $\mathtt{e}_1, \mathtt{e}_2 \in E$ *are* locally conflicting *iff 1.* $(\mathtt{e}_1, \mathtt{e}_2) \in \mathtt{sthd}$*; 2.* $\mathtt{loc}(\mathtt{e}_1) = \mathtt{loc}(\mathtt{e}_2)$*; and 3. at least one of* $\mathtt{e}_1, \mathtt{e}_2$ *is a write event. A pre-execution* $\mathcal{G}$ *is* locally data-race free *(LDRF), iff for all* $\mathtt{e}_1, \mathtt{e}_2 \in \mathcal{G}.E$*, if* $\mathtt{e}_1, \mathtt{e}_2$ *are locally conflicting, then* $(\mathtt{e}_1, \mathtt{e}_2) \in \mathcal{G}.\mathsf{gb} \cup \mathcal{G}.\mathsf{gb}^{-1}$*. Put differently, given the definition of* gb *(Fig. 5), a pre-execution* $\langle E, \mathsf{po}, \mathsf{pf}\rangle$ *is LDRF iff for all locally conflicting accesses* $\mathtt{e}_1, \mathtt{e}_2 \in E$*, if* $(\mathtt{e}_1, \mathtt{e}_2) \in \mathsf{po}$*, then the following four conditions hold:*

1. *If* $\mathtt{e}_1 \in \mathtt{nlW}$ *and* $(\mathtt{e}_1, \mathtt{e}_2) \notin \mathtt{sqp}$*, then there exists* $\mathtt{e}_3 \in \mathtt{P}$ *such that* $(\mathtt{e}_1, \mathtt{e}_3) \in \mathsf{pf}$ *and* $(\mathtt{e}_3, \mathtt{e}_2) \in \mathsf{po}$ *(cells E1, E2, and E5 in Fig. 5).*
2. *If* $\mathtt{e}_1 \in \mathtt{nlW}$*,* $\mathtt{e}_2 \in \mathtt{nlR}$*, and* $(\mathtt{e}_1, \mathtt{e}_2) \in \mathtt{sqp}$*, then either there exists* $\mathtt{e}_3 \in \mathtt{P}$ *with* $(\mathtt{e}_1, \mathtt{e}_3) \in \mathsf{pf}$ *and* $(\mathtt{e}_3, \mathtt{e}_2) \in \mathsf{po}$*; or there exists* $\mathtt{e}_3 \in \mathtt{nF}$ *with* $(\mathtt{e}_1, \mathtt{e}_3) \in \mathsf{po}$ *and* $(\mathtt{e}_3, \mathtt{e}_2) \in \mathsf{po}$ *(E6).*
3. *If* $\mathtt{e}_1 \in \mathtt{nlR}$*,* $\mathtt{e}_2 \in (\mathtt{nlW} \cup \mathtt{lW} \cup \mathtt{CAS})$*, and* $(\mathtt{e}_1, \mathtt{e}_2) \notin \mathtt{sqp}$*, then there exists* $\mathtt{e}_1' \in \mathtt{nrW}$ *and* $\mathtt{e}_3 \in \mathtt{P}$ *such that* $(\mathtt{e}_1, \mathtt{e}_1') \in \mathsf{po}|_{imm}$*,* $(\mathtt{e}_1', \mathtt{e}_3) \in \mathsf{pf}$*, and* $(\mathtt{e}_3, \mathtt{e}_2) \in \mathsf{po}$ *(cells B1 and B5).*
4. *If* $\mathtt{e}_1 \in \mathtt{nrR}$ *and* $\mathtt{e}_2 \in \mathtt{nrW}$*, then either there exists* $\mathtt{e}_3 \in \mathtt{nF}$ *such that* $\mathtt{e}_1 \xrightarrow{\mathsf{po} \cap \mathtt{sqp}} \mathtt{e}_3 \xrightarrow{\mathsf{po} \cap \mathtt{sqp}} \mathtt{e}_2$*; or there exists* $\mathtt{e}_1' \in \mathtt{nlW}$ *and* $\mathtt{e}_3 \in \mathtt{P}$ *such that* $\mathtt{e}_1 \xrightarrow{\mathsf{po}|_{imm}} \mathtt{e}_1' \xrightarrow{\mathsf{pf}} \mathtt{e}_3 \xrightarrow{\mathsf{po}} \mathtt{e}_2$ *(cell D7 in Fig. 5).*

These cases prohibit all possible races on a location x, i.e. of the form $x := y^n; x := -$ (E1,E5), $x := y^n; - := x$ (E2), $x := y^n; z^n := x$ (E6), $y^n := x; x := -$ (B1,B5), or $- := x^n; x^n := -$ (D7). Other entries in Fig. 5 cannot create races as either their ordering is already guaranteed (e.g. ✓ in E9); or they are on two read events (e.g. B2,D8); or they cannot be on the same location (e.g. D3,E7).

We argue that the constraints in Def. 7 do not restrict RDMA capabilities in that waiting for remote operations to complete before reusing their locations is already considered standard practice when writing RDMA programs.

We next show that LDRF prevents single-threaded weak behaviours.

Theorem 3. *Given a* $\textsc{rdma}^{sc}$*-consistent execution* $G = \langle E, \mathsf{po}, \mathsf{pf}, \mathsf{rf}, \mathsf{mo}, \mathsf{nfo}\rangle$*, if* $\langle E, \mathsf{po}, \mathsf{pf}\rangle$ *is locally data-race free, then there is no* sc *cycle on a single thread; that is,* $(G.\mathsf{sc} \cap \mathsf{sthd})$ *is acyclic and the first case of Theorem 2 does not arise.*

Preventing sc cycles across threads. Unlike data races, pairs of the second kind cannot create weak behaviours by themselves, and their reorderings can only be observed in certain contexts.

The general strategy to prevent observable reorderings is straightforward: for every pair $(\mathtt{e}_1, \mathtt{e}_2) \in \mathsf{po}$ on public locations, either we know for certain that $\mathtt{e}_2 \xrightarrow{\mathsf{sc}}{}^{*} \mathtt{e}_1$ (using other threads) is impossible, or we conservatively block the reordering by enforcing $(\mathtt{e}_1, \mathtt{e}_2) \in \mathsf{gb}$. The challenge is that the relation sc is heavily dependent on the specific execution. So how can we ascertain *syntactically* that a later event $\mathtt{e}_2$ cannot influence an earlier event $\mathtt{e}_1$?

One easily accessible syntactic property is the communication pattern between nodes (i.e. when one node performs a remote operation towards another).

Thus, to simplify the task, we over-approximate dependency (i.e. sc) with *communication*. Intuitively, if two nodes do not communicate in the network topology, then they cannot causally influence each other.

We write $n_1 \underset{E}{\rightsquigarrow} n_2$ (defined below) to denote that nodes n_1 and n_2 communicate via some event in E, in that some thread t on n_1 performs a remote operation $\mathsf{e} \in E$ towards n_2, written $\mathsf{hasQP}(t, n_2, E)$, or vice versa.

$$n_1 \underset{E}{\rightsquigarrow} n_2 \triangleq \exists t.\ (n(t) = n_1 \wedge \mathsf{hasQP}(t, n_2, E)) \vee (n(t) = n_2 \wedge \mathsf{hasQP}(t, n_1, E))$$

$$\mathsf{hasQP}(t, \overline{n}, E) \triangleq \exists \mathsf{e} \in (E.\mathtt{nrW} \cup E.\mathtt{nrR}).\ t(\mathsf{e}) = t \wedge \overline{n}(\mathsf{e}) = \overline{n}$$

We next show that if there is an sc-path from one event e_2 to another e_1 using public events in A, then the corresponding nodes (of the locations) of e_2 and e_1 must communicate via A. This is established in Lem. 1 below, with the proof given in the extended version [11].

Lemma 1. *For all* $A \subseteq E^{\mathsf{pub}}$*, if* $\mathsf{e}_2 \xrightarrow{\mathsf{sc}|_A}{}^{*} \mathsf{e}_1$ *then* $n(\mathtt{loc}(\mathsf{e}_2)) \underset{A}{\rightsquigarrow}^{*} n(\mathtt{loc}(\mathsf{e}_1))$.

We are interested in the inverse direction of this lemma: a topological connection between the nodes (of the locations) of e_2 and e_1 is a necessary condition for an sc-path from e_2 to e_1. Put differently, if there is no communication between the nodes of e_2 and e_1, then e_2 cannot influence e_1. As such, we can use this to over-approximate safely whether an event can influence another. We conservatively assume that if the two nodes can communicate (outside of the thread) then e_2 might influence e_1. These communications do not depend on a specific execution and can be ascertained syntactically from the pre-execution.

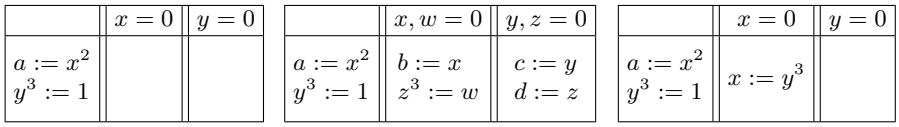
We can then prevent sc cycles across threads using the *fenced* condition below (Def. 8): for all $\mathsf{e}_1 \xrightarrow{\mathsf{po}} \mathsf{e}_2$ on public locations, if e_2 might influence e_1, then we block the reordering. We subsequently prove that if a pre-execution is fenced, then it does not admit sc cycles across threads.

Definition 8 (fenced). *A pre-execution* $\langle E, \mathsf{po}, \mathsf{pf}\rangle$ *is* fenced *iff for all* $\mathsf{e}_1, \mathsf{e}_2 \in E^{\mathsf{pub}}$*, if* $\mathsf{e}_1 \xrightarrow{\mathsf{po}} \mathsf{e}_2$ *and* $n(\mathtt{loc}(\mathsf{e}_1)) \underset{E^{\mathsf{pub}} \setminus t(\mathsf{e}_1)}{\rightsquigarrow}{}^{*} n(\mathtt{loc}(\mathsf{e}_2))$*, then* $(\mathsf{e}_1, \mathsf{e}_2) \in \mathsf{gb}$.

Theorem 4. *Given an* RDMA^{SC}*-consistent execution* $\langle E, \mathsf{po}, \mathsf{pf}, \mathsf{rf}, \mathsf{mo}, \mathsf{nfo}\rangle$*, if its associated pre-execution* $\langle E, \mathsf{po}, \mathsf{pf}\rangle$ *is fenced, then there is no* sc *cycle of the shape* $\mathsf{e}_1 \xrightarrow{\mathsf{po}\setminus\mathsf{ob}} \mathsf{e}_2 \xrightarrow{(\mathsf{sc};[E^{\mathsf{pub}}\setminus t(\mathsf{e}_1)])^{+};\mathsf{sc}} \mathsf{e}_1$ *with* $\mathsf{e}_1, \mathsf{e}_2 \in E^{\mathsf{pub}}$*. That is, the second case of Theorem 2 does not arise.*

Robustness. Lastly, we show that LDRF and fenced imply robustness under RDMA$^{\mathrm{SC}}$. Thus, this approach can be used to prevent RDMA weak behaviours.

Theorem 5 (Robustness under RDMA$^{\mathrm{SC}}$). *Given a pre-execution* $\mathcal{G} = \langle E, \mathsf{po}, \mathsf{pf}\rangle$*, if* $\mathcal{G}$ *is locally data-race free (Def. 7) and fenced (Def. 8), then* $\mathcal{G}$ *is also robust under* RDMA^{SC} *(Def. 5).*



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

Fig. 6: Examples illustrating the limitation of Theorem 5, where the programs in (a) and (b) are robust (the weak behaviour $a = 1$ is not allowed in either) while that in (c) is not robust (it admits the weak behaviour $a = 1$); while Theorem 5 rightfully identifies (a) as robust (true positive) and (c) as not robust (true negative), it conservatively deems (b) not robust (false negative).

4.3 Usage and Limitations

Local data-race freedom (Def. 7) and fenced (Def. 8) are intuitive properties that can be checked syntactically. Indeed, given a program, it is straightforward to check mechanically whether these properties hold or to provide an explicit counterexample and a suggested fix using the definition of gb (Def. 6). As a result, sufficient constraints can automatically be added to ensure robustness.

However, this simplicity can occasionally be the limitation of our approach. Specifically, as the main theorem does not account for interactions between threads, it takes a conservative approach, which at times can lead to false negatives (where the program is deemed not robust even though no weak behaviours are possible), recommending unnecessary restrictions.

To see this, consider the example in Fig. 6a, where $a := x^2$ and $y^3 := 1$ can be reordered *without* introducing weak behaviours. In this case, Theorem 5 rightfully confirms that no additional restrictions are necessary. By contrast, consider the variant shown in Fig. 6b: although the two extended threads do not introduce any additional weak behaviours, our approach assumes there might be a causal dependency from $y^3 := 1$ to $a := x^2$, as is the case e.g. in Fig. 6c. As such, Theorem 5 cannot determine Fig. 6b as robust, and our approach would recommend inserting a poll operation in the first thread. Note that removing any of the six operations would enable Theorem 5 to ascertain Fig. 6b as robust.

Understanding that the reordering of the instructions in the first thread of Fig. 6b is not problematic would require a more complex static analysis beyond the scope of this paper.

5 Application: Tree Topology

Theorem 5 outlines the conditions under which we can guarantee that a program is robust under RDMA$^{\text{SC}}$. However, while the LDRF property (Def. 7) is reasonable, the fenced property (Def. 8) can lead to excessive restrictions (e.g. as in Fig. 6). Specifically, for every pair of events $(\mathsf{e}_1, \mathsf{e}_2)$ in program order, we must either verify that e_2 cannot affect e_1, or ensure that their execution order is preserved. The main issue is that preserving the order of every pair of events can be particularly costly, notably when considering NIC remote write events. In

such cases, the only resort is to introduce a 'dummy' get operation and poll it, which is inefficient. Instead, we propose a strategy whereby we stipulate certain conditions on the network *topology* (i.e. the shape of the RDMA network) so that later events are often unable to influence earlier events.

To this end, we propose a *tree topology* that balances generality (supporting a wide range of programs) with efficiency and restrictiveness (requiring minimal additional constraints to respect the fenced property). In §5.1 we present an overview of our new set of restrictions and illustrate their rationale through examples. In §5.2 we formalises these restrictions and prove that they indeed imply robustness under RDMA$^{\text{SC}}$. Finally, in §5.3 we demonstrate specific applications of the tree topology and how RDMA programs can make use of them.

5.1 Overview of the Restrictions

We describe four different conditions that, if satisfied, ensure the robustness of RDMA programs under RDMA$^{\text{SC}}$, and we justify them through examples.

LDRF. As before, we require that programs satisfy LDRF (Def. 7). As discussed, this is considered standard practice when writing RDMA programs and should not be seen as a limitation.

Private Copies. We require the *local locations* of RDMA operations – e.g. location y in $y := x^2$ – to be private (i.e. accessed by only one thread, namely that executing the RDMA operation). Intuitively, to maximise the efficiency of RDMA programs, we should ideally allow arbitrary interleaving of RDMA operations and CPU computations. For instance, let us consider the single-threaded program $\mathsf{C} \triangleq y := x^2; \mathsf{c}^{\#}_{\mathsf{cpu}}$, where $\mathsf{c}^{\#}_{\mathsf{cpu}}$ denotes a block of CPU instructions that does not access location y. If y is private, then although $\mathsf{c}^{\#}_{\mathsf{cpu}}$ and the get $y := x^2$ may be reordered, this reordering will not lead to any observable weak behaviours. That is, when we run C concurrently with *any* RDMA program C' (i.e. as $\mathsf{C}||\mathsf{C}'$), if y is private, then we do not need to poll $y := x^2$ before proceeding with $\mathsf{c}^{\#}_{\mathsf{cpu}}$ (even though they may be reordered), as the reordering cannot be observed by C'.

However, if y is accessible by other threads (on the same node or from a remote node), then the reordering becomes visible, allowing additional, potentially unwanted, weak behaviours. This is illustrated in the example below, where $\mathsf{c}^{\#}_{\mathsf{cpu}} \triangleq z := 1$ and y is public (accessed by nodes 1 and 3).

$y, z = 0$	$x = 1$	
$y := x^2$ $z := 1$		$a := z^1$ $\texttt{poll}(1)$ $b := y^1$

$(a, b) = (1, 0)$ ✓

More concretely, due to the reordering, the later CPU computation ($z := 1$) can be observed before the earlier get ($y := x^2$), leading to the weak outcome $(a, b) = (1, 0)$.

Therefore, to prevent such weak behaviours, we stipulate that local locations of RDMA operations be private. This is not a costly limitation. Specifically, in the case of put operations, the data can easily be copied beforehand to a

one-time-use private location. In the case of get operations, it means the thread running the command needs to acknowledge the data and copy it to make it available to other threads having access to the node.

Get in Order. We stipulate that each get operation be followed by a remote fence. Recall that only certain reorderings are allowed on the operations of the same queue pair. Intuitively, put operations cannot be overtaken, and we do not need to restrict their usage. However, get operations can be overtaken by other get/put operations, as shown in the examples below, where the $a := x^2$ is overtaken by a later remote operation on the same queue pair, leading to weak behaviours.

	$x = 0$
$a := x^2$ $b := x^2$	$x := 1$

$(a, b) = (1, 0)$ ✓

	$x, y = 0$
$a := x^2$ $y^2 := 1$	$x := y$

$a = 1$ ✓

As such, to prevent non-SC behaviours, we require that each get operation be followed by a remote fence, forcing the queue pair to await the completion of the get before starting the next remote operation. Of course, if the get is polled before another RDMA operation is submitted, the remote fence is not needed. Note that since remote fences do not block CPU computations nor communications with other nodes, they are not very expensive and are a reasonable cost to pay to ensure remote operations towards a specific remote node stay in order.

Tree Topology. Finally, the most important restriction is to constrain the topology of the network over which the program runs. Intuitively, having multiple paths between a set of nodes allows for visible effects to overtake each other (i.e. be reordered) along different paths, leading to weak behaviours. In the extreme case where every thread can communicate directly with every other node, we allow for a large number of visible reorderings, and lose any hope of preventing non-SC behaviours. When such connected topologies are needed to enable more efficient implementations (e.g. consensus algorithms), the developers must carefully account for the possible weak behaviours.

Our proposal is to adhere to a minimal topology where there is (at most) a single communication path between each pair of nodes. In the examples below we show how not adhering to the tree topology can lead to weak behaviours. Note that although we have followed each remote operation with a corresponding (costly) poll, we still cannot prevent the weak behaviours shown.

$y = 0$	$x = 0$
$x^2 := 1$ `poll`(2) $a := y$	$y^1 := 1$ `poll`(1) $b := x$

$(a, b) = (0, 0)$ ✓

	$x = 0$	$y, z = 0$
$z^3 := 1$ `poll`(3) $x^2 := 1$	$y^3 := x$	$a := y$ $b := z$

$(a, b) = (1, 0)$ ✓

$x = 0$		$y, z = 0$
$z^2 := 1$ `poll`(2) $x := 1$	$y^2 := x$	$a := y$ $b := z$

$(a, b) = (1, 0)$ ✓

The first example shows that queue pairs in both directions (between nodes 1 and 2) can lead to weak behaviours as they can observe the reordering of

operations on the other node. The second example illustrates two paths between node 1 and 3: a direct path from node 1 to 3 (via $z^3 := 1$) and an indirect path through node 2 (from node 1 to 2 via $x^2 := 1$; from node 2 to 3 via $y^3 := x$). As shown, having multiple paths between two nodes allows threads to observe reorderings: $z^3 := 1$ is submitted first, but the effects of $x^2 := 1$, forwarded via $y^3 := x$, is observed first. The third example is a variant of the second, where the middle node is replaced by an additional thread on the left node. As queue pairs from different threads of the same node towards the same remote are still independent, the weak behaviour shown is permitted.

5.2 Tree Robustness

We next formalise the conditions described in §5.1 in Def. 9 below.

Definition 9 (tree-fenced). *A pre-execution* $\langle E, \mathsf{po}, \mathsf{pf}\rangle$ *is* tree-fenced *iff:*

1. *Local locations of RDMA operations are private:* $E^{\mathsf{pub}}.\mathtt{nlR} = E^{\mathsf{pub}}.\mathtt{nlW} = \emptyset$
2. *Each get operation is followed by a remote fence (or is polled) before the next remote operation on the same queue pair.*
   That is, for all $\mathsf{e}_1, \mathsf{e}_2$*, if* $\mathsf{e}_1 \in \mathtt{nrR}$*,* $\mathsf{e}_2 \in (\mathtt{nrR} \cup \mathtt{nrW})$*, and* $(\mathsf{e}_1, \mathsf{e}_2) \in (\mathsf{po} \cap \mathtt{sqp})$*, then: either there exists* $f \in \mathtt{nF}$ *such that* $(\mathsf{e}_1, f) \in (\mathsf{po} \cap \mathtt{sqp})$ *and* $(f, \mathsf{e}_2) \in (\mathsf{po} \cap \mathtt{sqp})$*; or there exists* $\mathsf{e}_3 \in \mathtt{nlW}$ *and* $p \in \mathtt{P}$ *such that* $(\mathsf{e}_1, \mathsf{e}_3) \in \mathsf{po}|_{imm}$*,* $(\mathsf{e}_3, p) \in \mathsf{pf}$*, and* $(p, \mathsf{e}_2) \in \mathsf{po}$*.*
3. *There is (at most) a single communication path between any pair of nodes in that the following three properties hold:*
   (a) *The network does not have cycles, i.e. for all sets of distinct nodes* $\{n_1; \ldots; n_k\}$ *with* $k > 2$*:* $\neg(n_1 \underset{E}{\rightsquigarrow} n_2 \underset{E}{\rightsquigarrow} \ldots \underset{E}{\rightsquigarrow} n_k \underset{E}{\rightsquigarrow} n_1)$
   (b) *No two nodes have queue pairs towards each other:* $\neg\exists t_1, t_2.\ \mathsf{hasQP}(t_1, n(t_2), E) \wedge \mathsf{hasQP}(t_2, n(t_1), E)$
   (c) *Each node has at most one queue pair towards each remote node:* $\forall t, t', \overline{n}.\ t \neq t' \wedge \mathsf{hasQP}(t, \overline{n}, E) \wedge \mathsf{hasQP}(t', \overline{n}, E) \implies n(t) \neq n(t')$

Conditions 1 and 2 are purely syntactic and can be straightforwardly checked by examining the RDMA program. Condition 3 pertains to the topology of the RDMA network and can also be checked by examining the RDMA program.

A key advantage of these restrictions is that preventing weak behaviours never requires polling remote operations. This is crucial because the efficiency of RDMA implementations comes from parallelising data transfers and computations. As shown in the overview (§2), polling is very costly as it completely halts local computations and prevents submission of remote operations to any queue pair. With a tree topology, programmers only need to wait for remote operations to use their results (as per LDRF Def. 7), and do not need to sacrifice computation time to prevent reorderings.

We next prove that if a pre-execution is tree-fenced, then it is also fenced. The full proof is given in the extended version [11].

Theorem 6. *If a pre-execution is tree-fenced (Def. 9), then it is fenced (Def. 8).*

Hence, LDRF and tree-fenced properties imply robustness under RDMA^{SC}.

Corollary 1 (Tree robustness under RDMA^{SC}). *If a pre-execution $\mathcal{G} = \langle E, \mathsf{po}, \mathsf{pf} \rangle$ satisfies LDRF (Def. 7) and is tree-fenced (Def. 9), then it is also robust under RDMA^{SC} (Def. 5).*

5.3 Specific Applications

The tree-fenced conditions above provide guidelines to ensure programs cannot exhibit weak behaviours. While not all RDMA programs follow the restrictions presented, a tree topology is sufficient for a range of applications. Notably, any setup using RDMA solely for the data transfer capabilities (and not for distributed computations) can easily be configured as a tree.

Star Topology: Single Manager Multiple Workers. The star topology is one of the most typical network configurations, providing simple and reliable communication between nodes, with many common applications such as for implementing local area networks (LAN). The star topology allows a main node to distribute jobs to other nodes and periodically check for progress. As demonstrated in this paper, this setup prevents any network weak behaviour even if communications towards different workers are independent and can be reordered.

Star Topology: Single Server Multiple Clients. The tree-fenced condition (Def. 9) is permissive enough to allow us to translate common concurrent algorithms (comprising loads and stores over shared memory) to distributed ones over RDMA *automatically* as follows. Specifically, consider a concurrent algorithm P_c using k threads $(t_1, ..., t_k)$. We can translate this to a corresponding RDMA program P_r using k nodes $(n_1, ..., n_k)$, where a designated node (say n_1) is the *server* and the others $(n_2, ..., n_k)$ are *clients*, and each node n_i has a single thread simulating t_i. All shared locations and data are located on the server node (n_1 running t_1). For each of the remaining nodes n_i, we replace the loads and stores on shared locations with get and put operations, respectively. Moreover, we insert a remote fence after each get operation (to ensure condition (2) of Def. 9) and poll get operations before using their values (to ensure LDRF).

The resulting RDMA program follows a *star topology*, with n_1 as the central (server) node accessed by multiple clients $(n_2, ..., n_k)$. Client locations are private by definition, ensuring that the tree-fenced condition holds. P_r thus avoids weak behaviours and constitutes a suitable implementation of P_c.

Observe that in this implementation, polling put operations is unnecessary (as long as different local locations are used for copying), and get operations can be optimised by being submitted as early as possible (i.e. after previous RDMA operations and reads on the same location) and before they are needed, allowing them to be interleaved with other computations.

6 Related Work

RDMA Semantics. The first realistic formal model for RDMA programs is RDMA^{TSO} by Ambal et al. [10] (where they assume that CPU concurrency is

governed by TSO) formalised both operationally and declaratively, which they show to be equivalent. They also validate RDMA$^{\text{TSO}}$ empirically by running an extensive suite of litmus tests on RDMA hardware. While comprehensive in its formal description of the language, this work does not present strategies for mitigating RDMA weaknesses or optimising the use of this technology by using e.g. minimal poll and fence instructions. The only other work on formal RDMA semantics is that by Dan et al. [27], which as demonstrated by Ambal et al. [10] does not follow the RDMA specification.

Weak Memory Models. Existing literature includes multiple examples of weak consistency models. For hardware, several works have formalised the semantics of the x86, ARMv8 and POWER architectures [68,9,2,63,48,5,59,31,67]. However, none of these works covered the consistency semantics of RDMA programs. For software, there has been a number of formal models for C11 [42,40,12,37,44,53,56,25] with verified compilation schemes [58,57,51], Java [49,15], transactional memory [72,61,60], the Linux kernel [8] and the ext4 filesystem [39]. Additionally, there has been several works on formalising the *persistency* semantics of programs in the context of non-volatile memory, describing the behaviour of programs in case of crashes [66,65,64,26,38], as well as program logics for verifying such programs [62,17,70].

Robustness. The concept of robustness against weak memory semantics has been extensively studied across various models as a means to simplify programming, reasoning, and verification. Notably, robustness for Total Store Order (TSO) and its Partial Store Order (PSO) variant [36,55,9] has received significant attention, e.g. [23,24,54,20,35,47,18,1,2,19,45,46]. In addition, robustness has been used as a correctness notion in the context of automatic fence insertion for weak hardware memory models [7,29,28,22,6]. More recent work has developed techniques for checking robustness against concurrency semantics in *programming languages*, particularly the C11 memory model [41,50]. Robustness has also been explored in distributed systems, where Sequential Consistency (SC) is replaced by *serialisability* [30,16,52,21,14,13]. More recently, [34] addressed the problem of checking robustness in the context of weak persistency models for non-volatile memory.

Some of these works provide sound and complete techniques for verifying robustness, along with complexity bounds for specific models. Others, as with our work on RDMA, focus on practical over-approximations, offering programmers guidelines that, when followed, ensure stronger semantics. The well-known Data-Race-Free (DRF) guarantee [3,33] for multicore hardware and programming language models is a prominent criterion of this type.

Acknowledgements. We thank the anonymous reviewers for their valuable feedback and Viktor Vafeiadis for many fruitful discussions. Ambal is supported by the EPSRC grant EP/X037029/1. Lahav is supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811) and the Israel Science Foundation (grant no. 814/22). Raad is supported by a UKRI fellowship MR/V024299/1, by the EPSRC grant EP/X037029/1 and by VeTSS.

References

1. Abdulla, P.A., Atig, M.F., Lång, M., Ngo, T.P.: Precise and sound automatic fence insertion procedure under PSO. In: NETYS. pp. 32–47. Springer International Publishing, Cham (2015)
2. Abdulla, P.A., Atig, M.F., Ngo, T.P.: The best of both worlds: Trading efficiency and optimality in fence insertion for tso. In: Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032. pp. 308–332. Springer-Verlag New York, Inc., New York, NY, USA (2015). https://doi.org/10.1007/978-3-662-46669-8_13, http://dx.doi.org/10.1007/978-3-662-46669-8_13
3. Adve, S.V., Hill, M.D.: Weak ordering—a new definition. In: ISCA. pp. 2–14. ACM, New York (1990). https://doi.org/10.1145/325164.325100, http://doi.acm.org/10.1145/325164.325100
4. Aguilera, M.K., Ben-David, N., Guerraoui, R., Marathe, V.J., Zablotchi, I.: The impact of RDMA on agreement. In: Robinson, P., Ellen, F. (eds.) Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019. pp. 409–418. ACM (2019). https://doi.org/10.1145/3293611.3331601, https://doi.org/10.1145/3293611.3331601
5. Alglave, J., Deacon, W., Grisenthwaite, R., Hacquard, A., Maranget, L.: Armed cats: Formal concurrency modelling at arm. ACM Trans. Program. Lang. Syst. **43**(2), 8:1–8:54 (2021). https://doi.org/10.1145/3458926, https://doi.org/10.1145/3458926
6. Alglave, J., Kroening, D., Nimal, V., Poetzl, D.: Don't sit on the fence: a static analysis approach to automatic fence insertion. ACM Trans. Program. Lang. Syst. **39**(2), 6:1–6:38 (May 2017). https://doi.org/10.1145/2994593, http://doi.acm.org/10.1145/2994593
7. Alglave, J., Maranget, L.: Stability in weak memory models. In: CAV. pp. 50–66. Springer-Verlag, Berlin, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=2032305.2032311
8. Alglave, J., Maranget, L., McKenney, P.E., Parri, A., Stern, A.: Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel. SIGPLAN Not. **53**(2), 405–418 (Mar 2018). https://doi.org/10.1145/3296957.3177156, https://doi.org/10.1145/3296957.3177156
9. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2) (Jul 2014). https://doi.org/10.1145/2627752, https://doi.org/10.1145/2627752
10. 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) (Oct 2024). https://doi.org/10.1145/3689781, https://doi.org/10.1145/3689781
11. Ambal, G., Lahav, O., Raad, A.: Extended version (2025), https://www.soundandcomplete.org/papers/ESOP2025/RDMA/
12. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing c++ concurrency. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 55–66. POPL '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/1926385.1926394, http://doi.acm.org/10.1145/1926385.1926394

13. Beillahi, S.M., Bouajjani, A., Enea, C.: Checking robustness against snapshot isolation. In: Computer Aided Verification. pp. 286–304. Springer International Publishing, Cham (2019)
14. Beillahi, S.M., Bouajjani, A., Enea, C.: Robustness against transactional causal consistency. In: CONCUR 2019. vol. 140, pp. 30:1–30:18. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). https://doi.org/10.4230/LIPIcs.CONCUR.2019.30
15. Bender, J., Palsberg, J.: A formalization of java's concurrent access modes. Proc. ACM Program. Lang. **3**(OOPSLA) (Oct 2019). https://doi.org/10.1145/3360568, https://doi.org/10.1145/3360568
16. Bernardi, G., Gotsman, A.: Robustness against consistency models with atomic visibility. In: CONCUR. pp. 7:1–7:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.7, http://drops.dagstuhl.de/opus/volltexte/2016/6165
17. 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)
18. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: ESOP. pp. 533–553. Springer-Verlag, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_29, http://dx.doi.org/10.1007/978-3-642-37036-6_29
19. Bouajjani, A., Enea, C., Mutluergil, S.O., Tasiran, S.: Reasoning about TSO programs using reduction and abstraction. In: CAV. pp. 336–353. Springer, Cham (2018)
20. Bouajjani, A., Meyer, R., Möhlmann, E.: Deciding robustness against total store ordering. In: ICALP. pp. 428–440. Springer, Berlin, Heidelberg (2011)
21. Brutschy, L., Dimitrov, D., Müller, P., Vechev, M.: Static serializability analysis for causal consistency. In: PLDI. pp. 90–104. ACM, New York (2018). https://doi.org/10.1145/3192366.3192415, http://doi.acm.org/10.1145/3192366.3192415
22. Burckhardt, S., Alur, R., Martin, M.M.K.: CheckFence: Checking consistency of concurrent data types on relaxed memory models. In: PLDI. pp. 12–21. ACM, New York (2007). https://doi.org/10.1145/1250734.1250737, http://doi.acm.org/10.1145/1250734.1250737
23. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: CAV. pp. 107–120. Springer-Verlag, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_12, http://dx.doi.org/10.1007/978-3-540-70545-1_12
24. Burnim, J., Sen, K., Stergiou, C.: Sound and complete monitoring of sequential consistency for relaxed memory models. In: TACAS. pp. 11–25. Springer, Berlin, Heidelberg (2011)
25. Chakraborty, S., Vafeiadis, V.: Grounding thin-air reads with event structures. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019). https://doi.org/10.1145/3290383, https://doi.org/10.1145/3290383
26. Cho, K., Lee, S.H., Raad, A., Kang, J.: Revamping hardware persistency models: View-based and axiomatic persistency models for intel-x86 and armv8. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 16–31. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454027, https://doi.org/10.1145/3453483.3454027

27. 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
28. Derevenetc, E.: Robustness against relaxed memory models. Ph.D. thesis, University of Kaiserslautern (2015), http://kluedo.ub.uni-kl.de/frontdoor/index/index/docId/4074
29. Derevenetc, E., Meyer, R.: Robustness against Power is PSpace-complete. In: ICALP. pp. 158–170. Springer, Berlin, Heidelberg (2014)
30. Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Shasha, D.: Making snapshot isolation serializable. ACM Trans. Database Syst. **30**(2), 492–528 (Jun 2005). https://doi.org/10.1145/1071610.1071615, http://doi.acm.org/10.1145/1071610.1071615
31. Flur, S., Gray, K.E., Pulte, C., Sarkar, S., Sezgin, A., Maranget, L., Deacon, W., Sewell, P.: Modelling the armv8 architecture, operationally: Concurrency and isa. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 608–621. POPL '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837615, https://doi.org/10.1145/2837614.2837615
32. Gerstenberger, R., Besta, M., Hoefler, T.: Enabling highly scalable remote memory access programming with mpi-3 one sided. Commun. ACM **61**(10), 106–113 (sep 2018). https://doi.org/10.1145/3264413, https://doi.org/10.1145/3264413
33. Gharachorloo, K., Adve, S.V., Gupta, A., Hennessy, J.L., Hill, M.D.: Programming for different memory consistency models. Journal of Parallel and Distributed Computing **15**(4), 399 – 407 (1992). https://doi.org/https://doi.org/10.1016/0743-7315(92)90052-0, http://www.sciencedirect.com/science/article/pii/0743731592900520
34. Gorjiara, H., Luo, W., Lee, A., Xu, G.H., Demsky, B.: Checking robustness to weak persistency models. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 490–505. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523723, https://doi.org/10.1145/3519939.3523723
35. Gotsman, A., Musuvathi, M., Yang, H.: Show no weakness: sequentially consistent specifications of TSO libraries. In: DISC. pp. 31–45. Springer-Verlag, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33651-5_3, http://dx.doi.org/10.1007/978-3-642-33651-5_3
36. Inc., S.I.: The SPARC architecture manual (version 9). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1994)
37. Kang, J., Hur, C.K., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. SIGPLAN Not. **52**(1), 175–189 (Jan 2017). https://doi.org/10.1145/3093333.3009850, https://doi.org/10.1145/3093333.3009850
38. Khyzha, A., Lahav, O.: Taming x86-tso persistency. Proc. ACM Program. Lang. **5**(POPL) (Jan 2021). https://doi.org/10.1145/3434328, https://doi.org/10.1145/3434328
39. Kokologiannakis, M., Kaysin, I., Raad, A., Vafeiadis, V.: Persevere: Persistency semantics for verification under ext4. Proc. ACM Program. Lang. **5**(POPL) (jan 2021). https://doi.org/10.1145/3434324, https://doi.org/10.1145/3434324

40. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. SIGPLAN Not. **51**(1), 649–662 (Jan 2016). https://doi.org/10.1145/2914770.2837643, https://doi.org/10.1145/2914770.2837643
41. Lahav, O., Margalit, R.: Robustness against release/acquire semantics. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 126–141. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314604, https://doi.org/10.1145/3314221.3314604
42. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in c/c++11. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 618–632. PLDI 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3062341.3062352, https://doi.org/10.1145/3062341.3062352
43. 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
44. Lee, S.H., Cho, M., Podkopaev, A., Chakraborty, S., Hur, C.K., Lahav, O., Vafeiadis, V.: Promising 2.0: Global optimizations in relaxed memory concurrency. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 362–376. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385412.3386010, https://doi.org/10.1145/3385412.3386010
45. Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in relaxed memory systems. In: SPIN. pp. 144–160. Springer-Verlag, Berlin, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=2032692.2032707
46. Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in PSO memory systems. In: TACAS. pp. 339–353. Springer-Verlag, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_24, http://dx.doi.org/10.1007/978-3-642-36742-7_24
47. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M., Yahav, E.: Dynamic synthesis for relaxed memory models. In: PLDI. pp. 429–440. ACM, New York (2012). https://doi.org/10.1145/2254064.2254115, http://doi.acm.org/10.1145/2254064.2254115
48. Mador-Haim, S., Maranget, L., Sarkar, S., Memarian, K., Alglave, J., Owens, S., Alur, R., Martin, M.M.K., Sewell, P., Williams, D.: An axiomatic memory model for POWER multiprocessors. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. Lecture Notes in Computer Science, vol. 7358, pp. 495–512. Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_36
49. Manson, J., Pugh, W., Adve, S.V.: The java memory model. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 378–391. POPL '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1040305.1040336, https://doi.org/10.1145/1040305.1040336
50. Margalit, R., Lahav, O.: Verifying observational robustness against a c11-style memory model. Proc. ACM Program. Lang. **5**(POPL) (Jan 2021). https://doi.org/10.1145/3434285, https://doi.org/10.1145/3434285
51. Moiseenko, E., Podkopaev, A., Lahav, O., Melkonian, O., Vafeiadis, V.: Reconciling Event Structures with Modern Multiprocessors (Artifact). Dagstuhl Arti-

facts Series **6**(2), 4:1–4:3 (2020). https://doi.org/10.4230/DARTS.6.2.4, https://drops.dagstuhl.de/opus/volltexte/2020/13201
52. Nagar, K., Jagannathan, S.: Automated detection of serializability violations under weak consistency. In: CONCUR 2018. vol. 118, pp. 41:1–41:18. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/LIPIcs.CONCUR.2018.41, http://drops.dagstuhl.de/opus/volltexte/2018/9579
53. Nienhuis, K., Memarian, K., Sewell, P.: An operational semantics for c/c++11 concurrency. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. p. 111–128. OOPSLA 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2983990.2983997, https://doi.org/10.1145/2983990.2983997
54. Owens, S.: Reasoning about the implementation of concurrency abstractions on x86-TSO. In: ECOOP. pp. 478–503. Springer-Verlag, Berlin, Heidelberg (2010)
55. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: TPHOLs. pp. 391–407. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_27
56. Pichon-Pharabod, J., Sewell, P.: A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 622–633. POPL '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837616, https://doi.org/10.1145/2837614.2837616
57. Podkopaev, A., Lahav, O., Vafeiadis, V.: Promising Compilation to ARMv8 POP. In: Müller, P. (ed.) 31st European Conference on Object-Oriented Programming (ECOOP 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 74, pp. 22:1–22:28. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2017). https://doi.org/10.4230/LIPIcs.ECOOP.2017.22, http://drops.dagstuhl.de/opus/volltexte/2017/7266
58. Podkopaev, A., Lahav, O., Vafeiadis, V.: Bridging the gap between programming languages and hardware weak memory models. Proc. ACM Program. Lang. **3**(POPL), 69:1–69:31 (Jan 2019). https://doi.org/10.1145/3290382, http://doi.acm.org/10.1145/3290382
59. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying arm concurrency: Multicopy-atomic axiomatic and operational models for armv8. Proc. ACM Program. Lang. **2**(POPL), 19:1–19:29 (Dec 2018). https://doi.org/10.1145/3158107, http://doi.acm.org/10.1145/3158107
60. 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)
61. 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)
62. Raad, A., Lahav, O., Vafeiadis, V.: Persistent owicki-gries reasoning: A program logic for reasoning about persistent programs on intel-x86. Proc. ACM Program. Lang. **4**(OOPSLA) (nov 2020). https://doi.org/10.1145/3428219, https://doi.org/10.1145/3428219
63. Raad, A., Maranget, L., Vafeiadis, V.: Extending intel-x86 consistency and persistency: Formalising the semantics of intel-x86 memory types and non-temporal

stores. Proc. ACM Program. Lang. **6**(POPL) (jan 2022). https://doi.org/10.1145/3498683, https://doi.org/10.1145/3498683
64. Raad, A., Vafeiadis, V.: Persistence semantics for weak memory: Integrating epoch persistency with the tso memory model. Proc. ACM Program. Lang. **2**(OOPSLA), 137:1–137:27 (Oct 2018). https://doi.org/10.1145/3276507, http://doi.acm.org/10.1145/3276507
65. Raad, A., Wickerson, J., Neiger, G., Vafeiadis, V.: Persistency semantics of the intel-x86 architecture. Proc. ACM Program. Lang. **4**(POPL) (Dec 2020). https://doi.org/10.1145/3371079, https://doi.org/10.1145/3371079
66. Raad, A., Wickerson, J., Vafeiadis, V.: Weak persistency semantics from the ground up: Formalising the persistency semantics of armv8 and transactional models. Proc. ACM Program. Lang. **3**(OOPSLA), 135:1–135:27 (Oct 2019). https://doi.org/10.1145/3360561, http://doi.acm.org/10.1145/3360561
67. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding power multiprocessors. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 175–186. PLDI '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1993498.1993520, https://doi.org/10.1145/1993498.1993520
68. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (Jul 2010). https://doi.org/10.1145/1785414.1785443, http://doi.acm.org/10.1145/1785414.1785443
69. Shpiner, A., Zahavi, E., Dahley, O., Barnea, A., Damsker, R., Yekelis, G., Zus, M., Kuta, E., Baram, D.: Roce rocks without pfc: Detailed evaluation. In: Proceedings of the Workshop on Kernel-Bypass Networks. p. 25–30. KBNets '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3098583.3098588, https://doi.org/10.1145/3098583.3098588
70. Vindum, S.F., Birkedal, L.: Spirea: A mechanized concurrent separation logic for weak persistent memory. Proc. ACM Program. Lang. **7**(OOPSLA2), 632–657 (2023). https://doi.org/10.1145/3622820, https://doi.org/10.1145/3622820
71. Wei, X., Shi, J., Chen, Y., Chen, R., Chen, H.: Fast in-memory transaction processing using rdma and htm. In: Proceedings of the 25th Symposium on Operating Systems Principles. p. 87–104. SOSP '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2815400.2815419, https://doi.org/10.1145/2815400.2815419
72. Xiong, S., Cerone, A., Raad, A., Gardner, P.: Data Consistency in Transactional Storage Systems: A Centralised Semantics. In: Hirschfeld, R., Pape, T. (eds.) 34th European Conference on Object-Oriented Programming (ECOOP 2020). Leibniz International Proceedings in Informatics (LIPIcs), vol. 166, pp. 21:1–21:31. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.21, https://drops.dagstuhl.de/opus/volltexte/2020/13178
73. 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. In: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication. p. 523–536. SIGCOMM '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2785956.2787484, https://doi.org/10.1145/2785956.2787484