Semantics of Remote Direct Memory Access: Operational and Declarative Models of RDMA on TSO Architectures

GUILLAUME AMBAL[*], Imperial College London, UK
BRIJESH DONGOL, University of Surrey , UK
HAGGAI ERAN, NVIDIA, Israel
VASILEIOS KLIMIS, Queen Mary University of London, UK
ORI LAHAV, Tel Aviv University, Israel
AZALEA RAAD, Imperial College London, UK

Remote direct memory access (RDMA) is a modern technology enabling networked machines to exchange information without involving the operating system of either side, and thus significantly speeding up data transfer in computer clusters. While RDMA is extensively used in practice and studied in various research papers, a formal underlying model specifying the allowed behaviours of concurrent RDMA programs running in modern multicore architectures is still missing. This paper aims to close this gap and provide semantic foundations of RDMA on x86-TSO machines. We propose three equivalent formal models, two operational models in different levels of abstraction and one declarative model, and prove that the three characterisations are equivalent. To gain confidence in the proposed semantics, the more concrete operational model has been reviewed by NVIDIA experts, a major vendor of RDMA systems, and we have empirically validated the declarative formalisation on various subtle litmus tests by extensive testing. We believe that this work is a necessary initial step for formally addressing RDMA-based systems by proposing language-level models, verifying their mapping to hardware, and developing reasoning techniques for concurrent RDMA programs.

CCS Concepts: • **Software and its engineering** → **Formal language definitions**; • **Theory of computation** → **Program semantics**; **Distributed computing models**; • **Hardware** → Testing with distributed and parallel systems.

Additional Key Words and Phrases: RDMA, Operational Semantics, Declarative Semantics, x86-TSO

ACM Reference Format:
Guillaume Ambal, Brijesh Dongol, Haggai Eran, Vasileios Klimis, Ori Lahav, and Azalea Raad. 2024. Semantics of Remote Direct Memory Access: Operational and Declarative Models of RDMA on TSO Architectures. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 341 (October 2024), 28 pages. https://doi.org/10.1145/3689781

1 Introduction

Remote direct memory access (RDMA) technologies such as *InfiniBand* and *RDMA over Converged Ethernet* (RoCE) enable a machine to have *direct* read/write access to the memory of another machine over a network, bypassing the operating systems of both machines. This way, remote reads and writes are performed with far fewer CPU cycles, leading to high-throughput, low-latency

[*] Corresponding author

Authors' Contact Information: Guillaume Ambal, Imperial College London, UK, g.ambal@imperial.ac.uk; Brijesh Dongol, University of Surrey , UK, b.dongol@surrey.ac.uk; Haggai Eran, NVIDIA, Israel, haggaie@nvidia.com; Vasileios Klimis, Queen Mary University of London, UK, v.klimis@qmul.ac.uk; Ori Lahav, Tel Aviv University, Israel, orilahav@tau.ac.il; Azalea Raad, Imperial College London, UK, azalea.raad@imperial.ac.uk.

ACM 2475-1421/2024/10-ART341
https://doi.org/10.1145/3689781

networking, which is especially useful in massively parallel computer clusters, e.g. for data centres, big data, and scientific computation. Thanks to implementations that offer higher performance at a comparable cost over traditional networking infrastructure (e.g. TCP/IP sockets) [Gerstenberger et al. 2018], RDMA has achieved widespread adoption as of 2018 [Shpiner et al. 2017] and has been rapidly adopted in modern data centres.

At the lowest level, RDMA networks directly interact with the hardware through calls to read (get) and write (put) operations to remote memory locations. As a result, programming RDMA systems is conceptually similar to shared memory systems of existing hardware architectures such as Intel-x86 or ARM. A key difference, however, is that when a machine encounters a remote operation, the CPU forwards it onto the *network interface card* (NIC), which subsequently handles the remote operation and its associated memory accesses without further CPU involvement.

There is a wide range of RDMA implementations, starting from the Virtual Interface Architecture (VIA) [Dunning et al. 1998], later adapted by InfiniBand [IBTA 2022] and RoCE [InfiniBand Trade Association (IBTA) 2018]. Other standards include iWarp [Recio et al. 2007] and Omni-Path [Birrittella et al. 2015]. Some transfer technologies without NICs, such as FireWire [Anderson 1999], can also be considered RDMA, but they do not scale to wide networks. In most implementations, an RDMA NIC implements the transport in hardware and is controlled by software through APIs such as Verbs [linux-rdma 2018] or libfabric [OpenFabrics 2016]. A NIC typically connects to the host CPU through an internal server fabric such as PCIe, though in some cases the NIC and compute cores can be more tightly integrated, e.g. in academic proposals [Novakovic et al. 2014] and in DPUs [NVIDIA Corporation 2021]. In addition, there have been proposals for next-gen fabrics to replace PCIe (e.g. CXL [Van Doren 2019]).

Our focus here is on the IB Verbs model defined by IBTA [2022], using PCIe as the internal fabric. It was designed for InfiniBand and reused for RoCE, the two most popular RDMA technologies. While the specification for the iWarp protocol is slightly more permissive, its main implementation (libfabric) follows the stronger semantics presented here. Lastly, the more recent Omni-Path fabric also has legacy support for IB Verbs.

The performance gains of RDMA, as well as its wide range of implementations, have led to a surge of both theoretical and practical RDMA research [Aguilera et al. 2019; Dan et al. 2016; Wei et al. 2015; Zhu et al. 2015]. However, as we discuss below, programming RDMA systems *correctly* is not straightforward, and the RDMA community would benefit greatly from formal models and rigorous techniques for reasoning about RDMA programs.

A key challenge lies in understanding the different degrees of concurrency and their interaction thereof. More concretely, a program may comprise threads that run over multiple nodes (machines) over the network (inter-node concurrency), with each node itself executing several threads (intra-node concurrency). As such, to understand the behaviour of a concurrent RDMA program, one must understand how remote and local operations on different nodes interact with one another. The problem is that local operations are handled by the CPU, while remote operations are handled by the NIC independently and in *parallel* to other CPU operations. Consequently, two sequential operations may not be executed in the intended (program) order, leading to surprising outcomes.

To understand the behaviour of RDMA programs, we must understand the order in which CPU and remote operations are executed, how they may be *reordered*, and how and when their effects are made visible to concurrent threads, be they on the same or different nodes. Specifically, much in the same way that the semantics of multi-processor hardware architectures such as Intel-x86, POWER, and ARM have been described via *formal consistency models* (a.k.a. *memory models*) [Alglave et al. 2021, 2014; Cho et al. 2021; Mador-Haim et al. 2012; Pulte et al. 2018, 2019; Raad et al. 2022; Raad and Vafeiadis 2018; Raad et al. 2020b, 2019b; Sarkar et al. 2011; Sewell et al. 2010], we should ideally describe the semantics of RDMA programs *formally*. Such formal models not only provide a rigorous underpinning for reasoning about the behaviour of programs, they have also been historically successful at identifying mistakes and ambiguities in the existing hardware reference manuals, as well as compilation bugs [Alglave et al. 2021, 2014; Lahav et al. 2017; Pulte et al. 2018; Raad et al. 2020b].

Unfortunately, the existing literature includes next to no work on the formal semantics of RDMA programs. Indeed, to our knowledge, the coreRMA model by Dan et al. [2016] is the only one to offer a formal description of RDMA programs. However, this work has four key shortcomings.

First, coreRMA assumes that CPU concurrency on each node is governed by the *sequential consistency* (SC) [Lamport 1979] model. This is an unrealistic assumption as no existing CPU architecture supports SC by default, and the two mainstream CPU architectures, Intel-x86 and ARM, are both subject to weaker models that exhibit behaviours not possible under SC. This is a significant gap since Intel-x86 and ARM architectures are ubiquitous.

Second, coreRMA describes the semantics *only declaratively* (i.e. via execution graphs that are subject to a set of axioms stipulating the absence of certain cycles in the graphs) and *not operationally* (via transitions that describe how the underlying hardware processes each operation and manipulates the memory). While declarative models are common in the literature of weak memory models and are more concise, operational models provide a more intuitive account of the hardware guarantees (as they prescribe a step-by-step mechanism for producing the behaviours of a program). Moreover, having two characterisations is useful not only for ensuring the accuracy of the formalism, but also because each formulation may be more useful for establishing different results. In particular, operational models are better suited for underpinning program logics and checking the reachability of an erroneous configuration and/or robustness for finite-state programs with loops (e.g. [Abdulla et al. 2021; Bouajjani et al. 2013; Lahav and Boker 2020]).

Third, the coreRMA authors have failed to *validate* their model against existing implementations in that they could not observe *any* of the weak behaviours allowed by coreRMA on existing implementations. That is, they could not practically justify the weakness of coreRMA.

Fourth and most importantly, as we discuss in detail in §6, coreRMA is not faithful to the RDMA specification [IBTA 2022] and departs from it in three different ways. In particular, coreRMA is neither stronger nor weaker than the specification, meaning that it admits certain behaviours disallowed by the specification, while prohibiting others allowed by the specification.

To close these gaps, we present RDMA$^{\text{TSO}}$, the *first* formal semantics of RDMA programs in the context of the x86 architecture, which implements the TSO model [Sewell et al. 2010]. We describe RDMA$^{\text{TSO}}$ both *operationally* and *declaratively* and prove that the two are *equivalent*. Specifically, we first develop *two operational models* of RDMA$^{\text{TSO}}$: (1) a *concrete* model, reflecting the hardware structure for propagating data across the network; and (2) a *simplified* model, abstracting away the hardware details, resulting in a cleaner model. We prove that our two operational characterisations of RDMA$^{\text{TSO}}$ are equivalent and mechanise our proof in Coq. We then present a declarative model of RDMA$^{\text{TSO}}$ and show that it is equivalent to our simplified (and thus also concrete) operational model.

We have developed RDMA$^{\text{TSO}}$ in close collaboration with engineers at NVIDIA, the largest manufacturer of networking products worldwide (after acquiring Mellanox in 2019). In particular, we have discussed all weak behaviours admitted by RDMA$^{\text{TSO}}$ with the engineers and have reflected the hardware justification for such behaviours in our concrete semantics. To further increase confidence in the fidelity of RDMA$^{\text{TSO}}$ to the specification, we have *empirically validated* it via extensive testing on existing implementations. More specifically, through our empirical validation we have managed to establish that (1) RDMA$^{\text{TSO}}$ is *not too strong*: we did not observe any of the behaviours prohibited by RDMA$^{\text{TSO}}$ on existing implementations; and (2) RDMA$^{\text{TSO}}$ is not *too weak*: we managed to observe *almost all* weak behaviours allowed by RDMA$^{\text{TSO}}$, on existing implementations, and in the few cases where we did not observe a weak behaviour allowed by RDMA$^{\text{TSO}}$, the engineers at NVIDIA

$x = y = 0$	
$x := 1$ $a := y$	$y := 1$ $b := x$

(a) $a = b = 0$ ✓

$x = y = 0$	
$x := 1$ `mfence` $a := y$	$y := 1$ `mfence` $b := x$

(b) $a = b = 0$ ✗

$x = y = 0$	
$a := y$ $x := 1$	$b := x$ $y := 1$

(c) $a = b = 1$ ✗

$x = y = 0$	
$x := 1$ $y := 1$	$a := y$ $b := x$

(d) $a = 1, b = 0$ ✗

Fig. 1. TSO litmus tests for CPU concurrency, where locations x, y are accessed by all threads, while locations a, b are accessed by one thread only, and $x = y = 0$ on the first line denotes that x, y initially hold value 0.

confirmed that current implementations explicitly do not utilise the weakness admitted by the RDMA specification (see §5).

Contributions and Outline. In §2 we present an intuitive account of RDMA$^{\text{TSO}}$ through a number of examples. In §3 we present our concrete and simplified operational semantics of RDMA$^{\text{TSO}}$ and show that they are equivalent. In §4 we present our declarative semantics of RDMA$^{\text{TSO}}$ and show that it is equivalent to our simplified operational semantics. In §5 we describe how we empirically validated RDMA$^{\text{TSO}}$ through extensive litmus testing. We discuss related and future work in §6.

Additional Material. The full proofs of all stated theorems are given in the extended version [Ambal et al. 2024a] and accompanying Coq development [Ambal et al. 2024b]. We provide the executable RDMA code (in machine-readable format) and detailed instructions for replicating our experiments and analysing our litmus tests [Ambal et al. 2024b].

2 Overview

We present an account of the formal RDMA semantics, RDMA$^{\text{TSO}}$, through a number of examples. We model concurrent programs running over a network of machines, and hereafter refer to each machine on the network as a *node*. In our setting, the semantics of a concurrent program and thus its possible weak behaviours are determined by two factors: (1) the *origin* of the threads, i.e. whether all threads originate from (are forked by) the same node and thus the concurrency is *intra-node* (within one node), or they originate from several nodes and thus concurrency is *inter-node* (across two or mode nodes); and (2) the *memory* targeted by the threads, i.e. whether each thread accesses its own local memory (on the same node), that of other nodes, or a combination thereof.

Litmus Test Outcome Notation. In the remainder of this article, we present small representative examples (known as litmus tests in the literature) to illustrate whether an outcome is allowed by a given model (e.g. in Fig. 1, Fig. 2 and Fig. 3), and annotate a given outcomes with: (1) ✓, to denote that the outcome is *allowed* by the model *and observed* in practice (in our empirical validation); (2) ✓*, to denote that the outcome is *allowed* by the model *and not observable* in practice; (3) ✗, to denote that the outcome is *disallowed* by the model *and not observed* in practice. See §5 or the extended version for more details.

CPU Concurrency and TSO. Existing work on RDMA semantics [Dan et al. 2016] assumes that CPU concurrency on each node is governed by the strong and unrealistic *sequential consistency* (SC) model [Lamport 1979]. We relax this assumption here and instead model each node as an x86 machine, subject to the TSO memory model [Sewell et al. 2010] introduced by the SPARC architecture [SPARC 1992]. Under TSO, a later read (in program order) can be reordered before an earlier write on a different location. This is illustrated in the *store buffering* example of Fig. 1a, where x and y denote locations accessed by both threads, while a and b denote locations accessed by

$x=0$	$z=0$
$x := 1$ $z^2 := x$	

(a) $z=0$ ✗ $z=1$ ✓

$x=0$	$z=0$
$z^2 := x$ $x := 1$	

(b) $z=0$ ✓ $z=1$ ✓

$x=0$	$z=0$
$z^2 := x$ `poll(2)` $x := 1$	

(c) $z=0$ ✓ $z=1$ ✗

$x=0$	$z=0$
$z^2 := x$ $z^2 := x$ `poll(2)` $x := 1$	

(d) $z=0$ ✓ $z=1$ ✓

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

(e) $x=1$ ✓ $x=2$ ✓

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

(f) $y=0$ ✓ $y=1$ ✓

$x=1$	$y=z=0$
$z^2 := x$ $x := y^2$	

(g) $z=0$ ✗ $z=1$ ✓

$x = 1$	$y=z=0$
$x := y^2$ $z^2 := x$	

(h) $z=0$ ✓ $z=1$ ✓

$x = 1$	$y=z=0$
$x := y^2$ `rfence (2)` $z^2 := x$	

(i) $z=0$ ✓ $z=1$ ✗

Fig. 2. Sequential RDMA litmus tests (excerpt), where each column (separated by ||) denotes a distinct node, the statement on the top line of each column denotes the initial values of locations, and the statements in the caption express whether each outcome is allowed by RDMA$^{\text{TSO}}$ and observed in practice (✓), or disallowed by RDMA$^{\text{TSO}}$ and not observed in practice (✗); i.e. we have empirically validated all outcomes shown.

one thread only.[1] Specifically, the reads $a := y$ and $b := x$ can respectively be reordered before the writes $x := 1$ and $y := 1$, allowing them to read the initial value 0 from y and x, yielding $a = b = 0$ at the end of execution. Note that this weak behaviour is not allowed under the stronger SC model as SC admits no instruction reordering.

To prevent such write-read reordering, one can use an `mfence` as in Fig. 1b: `mfence` instructions cannot be reordered in either direction, and thus the weak behaviour shown is no longer possible. Indeed, other than write-read reordering, TSO admits no other reorderings and thus other weak behaviours, e.g. *load buffering* in Fig. 1c and *message passing* in Fig. 1d are prohibited under TSO.

Remote Direct Memory Access (RDMA). RDMA allows one to build a network of communicating nodes (machines), where each node can *directly* access remote memory (of other nodes) through its network interface card (NIC). RDMA networks are programmed via operations that read from and write to remote memory, as well as various synchronisation operations. As such, programming RDMA networks is conceptually similar to shared memory systems such as TSO.

To distinguish remote (RDMA) operations from CPU ones, we refer to RDMA reads and writes as *get* and *put* operations, respectively. Moreover, to distinguish local and remote memory locations, we write x^n for a memory location on a remote node n, and write x for a memory location on the current local node. A put operation is of the form $x^n := y$, and consists of reading from a local memory location y (referred to as a 'NIC local read') and writing to a remote memory location x on node n (a 'NIC remote write' or simply a 'remote write'). Similarly, a get operation is of the form $x := y^n$, and consists of reading from a remote memory location y on node n (a 'NIC remote read' or a 'remote read') and writing to a local memory location x (a 'NIC local write'). We write $\overline{n}$ to identify a node other than n. When a thread on local node n issues a remote operation to be executed on remote node $\overline{n}$, we denote this by stating that the operation is *by n towards $\overline{n}$*.

Sequential (Single-Threaded) RDMA$^{\text{TSO}}$ Behaviours. When a thread issues a put or get operation, it is handled by the NIC subsystem (and its associated queue pairs and buffers as shown

[1] In our general model, all memory locations are shared and thus can be accessed by all threads both locally (on the same node) and remotely. However, for better readability, we follow the convention of naming locations accessed by multiple threads (locally or remotely) as x, y, z and w, while naming locations accessed by a single local thread as a, b, c and d.

in Fig. 4), in contrast to the CPU operations which are handled by the processor subsystem (and its associated store buffers). 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. 2, where each column represents a distinct node, numbered from 1 onwards. For instance, the example in Fig. 2a 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. 2a: as $z^2 := x$ follows $x := 1$, it observes the $x := 1$ write and thus puts value 1 to z; i.e. outcome $z = 0$ is not permitted.

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. 2b, where the earlier $z^2 := x$ may execute (be reordered) after the later $x := 1$, and thus both $z = 0$ and $z = 1$ outcomes are possible.

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

Note that each $\texttt{poll}(n)$ waits for *only one* (the earliest in program order) and *not all* pending remote operation towards n to complete. This allows for more efficient and fine-grained control over remote operations, but requires some care. For instance, consider the example in Fig. 2d, where $\texttt{poll}(2)$ blocks until the *first* $z^2 := x$ is complete, and thus the second $z^2 := x$ operation can be reordered after the later $x := 1$, once again allowing for the $z = 1$ outcome.

Two remote operations towards *different* nodes are fully independent and can execute in either order (as if within two separate threads). For instance, the two get operations in Fig. 2e can execute in either order, and thus the final value of x may be either that of z ($x = 1$ outcome) or that of y (the $x = 2$ outcome). Similarly, Fig. 2f has two possible outcomes. The only way to enforce the execution order is polling the first remote 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 remote (NIC) put operation $x^n := y$ comprises two steps: a NIC local read (obtaining the value of y) and a NIC remote write (writing the value of y to x^n). Similarly, a remote get operation $x := y^n$ comprises two steps: a NIC remote read (obtaining the value of y^n) and a NIC local write (writing the value of y^n to x). Intuitively, for remote operations on a given location x, these four steps mandate a *precedence* which in turn determines whether they can be reordered. Specifically, the four steps described above give way to the following precedence order: i) NIC local read; ii) NIC remote write; iii) NIC remote read; iv) NIC local write.

If a step with a higher precedence (e.g. a NIC local read) is in program order before one with a lower precedence (e.g. a NIC local write), then their order is preserved and they cannot be reordered; otherwise the order is not necessarily preserved and these steps can be reordered. For instance, in the Fig. 2g example, the earlier NIC local read on x (in $z^2 := x$) has a higher precedence than the

[2] In the Verbs API, $\texttt{poll}(n)$ returns a value denoting whether the operation being polled has completed and does not block the execution. However, it is common practice for calls to poll to be placed in a spin loop that returns only when the operation is completed, hence blocking the execution. Here, we model this common pattern with stronger behaviours.

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

(a) $a=b=0$ ✓

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

(b) $a=b=1$ ✓

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

(c) $a=b=1$ ✗

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

(d) $a=b=0$ ✓*

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

(e) $a=b=0$ ✗

Fig. 3. Concurrent RDMA litmus tests (excerpt). The annotations in the captions indicate the given outcome is allowed by RDMA$^{\text{TSO}}$ *and observed* in our empirical validation (✓), allowed by RDMA$^{\text{TSO}}$ *and not observable* in practice (✓*), or disallowed by RDMA$^{\text{TSO}}$ *and not observed* in our validation (✗). That is, we have empirically validated all outcomes shown, except that in (d), which is due to the underlying implementation explicitly not utilising the weakness admitted by the specification – see §5 or the extended version for more details.

later NIC local write on x (in $x := y^2$), and thus the order of these steps on x is preserved; i.e. the old value (1) of x is written to z^2 leading to the $z = 1$ outcome, and the $z = 0$ outcome is disallowed.

By contrast, in the Fig. 2h example, the earlier NIC local write on x (in $x := y^2$) has a lower precedence than the later NIC local read on x (in $z^2 := x$) and thus the two steps can be reordered. Besides the SC outcome ($z = 0$), the program might execute the NIC local read on x before the NIC local write on x, thereby reading the initial value 1 from x and writing it to z (outcome $z = 1$).

As before, the reordering of the two remote operations in Fig. 2h can be prevented by polling the first operation before the second operation. However, polling is costly as it leads to global blocking: it blocks the current thread *both* on the CPU and the NIC towards *all* nodes (i.e. the current thread cannot execute any remote operations on *any* node). Alternatively, we can use a *remote fence*[3]operation, `rfence` (n), that blocks the current thread *only* on the NIC and *only* towards node n. (i.e. the thread cannot execute any remote operations towards n, but can execute both on the CPU as well as on the NIC towards nodes other than n). This in turn ensures that earlier (in program order) remote operations by the thread towards n (those before the fence) are executed before later remote operations towards n (those after the fence). This is illustrated in Fig. 2i, obtained from Fig. 2h by inserting `rfence` (2) between the two remote operations towards node 2, thereby ensuring that they are executed in order, and thus $z = 1$ is no longer possible.

Concurrent (Multi-threaded) RDMA$^{\text{TSO}}$ Behaviours. The real power of RDMA comes from multiple programs running on different nodes. This introduces a wide range of weak behaviours, as we describe below. A network can comprise several nodes, each running several concurrent threads. Here, we focus on a few examples each with two nodes, with each node comprising a single thread.

The store buffering behaviour due to the TSO model discussed in Fig. 1a is also possible in the RDMA setting, i.e. when locations x and y are on two different nodes, as shown in Fig. 3a. Moreover, further weak behaviours not possible under TSO, e.g. load buffering in Fig. 1c, are permitted in the RDMA setting, as shown in Fig. 3b. As before, this can be intuitively justified by conceptually viewing the remote operations in each thread as being executed by a separately spawned thread.

Most weak behaviours such as load buffering in Fig. 3b can be prevented by polling the remote operations as needed, as shown in Fig. 3c. Specifically, the poll operations in Fig. 3c await the completion of the preceding get operations, and thus the earlier get operations cannot be reordered after the later writes, thus prohibiting the weak behaviour $a=b=1$. However, a notable exception to this is the store buffering weak behaviour which cannot be prevented even when polling the

[3] For InfiniBand Verbs, this remote fence is not an independent operation but a flag that can be set on the later operation.

remote operations, as shown in Fig. 3d. This is because the specification of polling offers different guarantees for get and put operations. Specifically, polling a get operation $a := x^2$ offers a strong guarantee and behaves intuitively: polling ensures that the get operation is complete (i.e. the value of x^2 is fetched from node 2), and the executing thread performs the associated NIC local write on a before marking the operation as complete and proceeding with the remainder of the execution. By contrast, polling a put operation $x^2 := 1$ offers a weaker guarantee: when sending value 1 towards node 2 to be put in x^2, the remote NIC merely *acknowledges* having received the data (value 1), but this data may still reside in a *buffer* (i.e. the PCIe fabric) of the remote node, *pending* to be written to the remote memory. Polling a put operation then simply awaits the acknowledgement of the data receipt and not its full completion (the data being written to memory). 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. 3d, it is possible for both poll operations to complete before the values of x and y are updated (to 1) in memory, and thus for the later reads to read their old values (0). However, note that existing implementations on current hardware (including ones against which we validated our model,) do not utilise this flexibility admitted by the specification. As such, the weak behaviour in Fig. 3d is not observable in practice, and thus we could not observe them in our validation effort (as indicated by ✓*) – see §5. Nevertheless, since our aim is to capture the *specification* and not the *implementation*, we have modelled RDMA$^{\text{TSO}}$ to allow this weak behaviour.

The behaviours discussed thus far all hold of the general RDMA specification [IBTA 2022] as well as the PCIe standard [PCI-SIG 2022]. However, in certain cases the PCIe standard offers stronger guarantees than those delineated by the RDMA specification. In particular, PCIe does not allow a read to fetch a pending value that has not yet been committed to memory. As such, a NIC remote read *flushes* (commits) all pending NIC remote writes to memory, while a NIC local read flushes all pending NIC local writes to memory. Interestingly, we can use this guarantee to prevent weak behaviours such as store buffering (which, in theory, cannot be prevented even via polling). Specifically, recall that polling a put only ensures that the data transmitted has reached the remote node and may not have yet been committed to its memory. However, by polling a later get (towards the same remote node) we can ensure the previous NIC remote writes (including that of the recently polled put) have been committed to the remote memory. An example of this is shown in Fig. 3e, obtained from Fig. 3d by adding additional gets ($c := z^2$ and $d := w^1$) and subsequently polling them. Reading from z^2 and w^2 in effect flushes the pending NIC remote writes on both nodes, ensuring that the effects of the earlier puts ($x^2 := 1$ and $y^1 := 1$) are committed to memory, which in turn ensures that the later $a := y$ and $b := x$ reads observe the updates on y and x (value 1), thus prohibiting $a = b = 0$.

As mentioned above, this guarantee is PCIe-specific, and not mentioned by the RDMA standard. However, as PCIe is the *de facto* standard for RDMA programming, and since all widely available RDMA hardware is PCIe-compatible, here we opt to model this guarantee, resulting in a stronger model. Nevertheless, in §3 and §4 we also describe how we can weaken our models by removing this guarantee, both in our operational and declarative semantics.

Our aim here was to provide an overview of the weak RDMA behaviours both in the sequential and concurrent settings. We refer the reader to the extended version [Ambal et al. 2024a] for further examples of weak behaviours.

3 RDMA$^{\text{TSO}}$ Concrete and Simplified Operational Semantics

We begin with several preliminary concepts. We present an informal account of our concrete semantics (§3.1), our formal concrete semantics (§3.2) and our equivalent simplified semantics (§3.3).

Nodes and Threads. We consider a system with N nodes (machines) and M threads in total across all machines. Let $\mathsf{Node} = \{1, \ldots, N\}$ be the set of *node identifiers*, and $\mathsf{Tid} = \{1, \ldots, M\}$ be the set of *thread identifiers*. 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 $n(t)$. Note that multiple threads may run on the same node.

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 and $x^{\overline{n}}, y^{\overline{n}}, z^{\overline{n}}$ to range over Loc_n and $\mathsf{Loc}_{\overline{n}}$, respectively. When the choice of n is clear, we write x for x^n; similarly for $\overline{n}$. For clarity, we use distinct location names across nodes, and thus write $n(x)$ for the unique $n \in \mathsf{Node}$ where $x \in \mathsf{Loc}_n$.

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 (as these are standard). We denote by Exp the set of all expressions and use e to range over Exp. We write $[[e]]$ for the semantic evaluation of a *closed* expression e (i.e. one without any locations), $\mathsf{elocs}(e)$ for the locations in an expression e, and $e[v/x]$ for the expression obtained from e after substituting all occurrences of x by v. We use e^n to range over expressions with $\mathsf{elocs}(e^n) \subseteq \mathsf{Loc}_n$.

Sequential Commands and Programs. *Sequential* programs running 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 for a finite, possibly zero, number of iterations). A (concurrent) *program*, P, is a map from thread identifiers to commands, associating each thread $t \in \mathsf{Tid}$ with a command on node $n(t)$.

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

Primitive commands include *CPU* (cc^n) and *RDMA* (rc^n) operations. A CPU operation on n may be a no-op (skip), an assignment to a local location ($x := e$), an assumption on the value of a local location ($\mathtt{assume}(x = v)$ and $\mathtt{assume}(x \neq v)$), a memory fence (mfence), 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) an 'RDMA 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 $\overline{n}$. Note that $\mathtt{poll}\ (\overline{n})$ is executed by the CPU and blocks its thread (and prevents the requests of later remote operations), while $\mathtt{rfence}\ (\overline{n})$ blocks the NIC for the execution of remote operations towards node $\overline{n}$. In what follows, we write $\overline{x} := 1$ as a shorthand for $\overline{x} := a$ for some local location a containing value 1.

3.1 RDMA$^{\text{TSO}}$ Concrete Operational Semantics at a Glance

RDMA$^{\text{TSO}}$ Architecture and CPU Operations. Conceptually, the RDMA$^{\text{TSO}}$ architecture is as shown at the top-right of Fig. 4, where for brevity we depict two nodes, each comprising (1) m threads and their associated FIFO (first-in-first-out) *store buffers*; (2) a network interface card (NIC) and its PCIe root complex; and (3) the memory. Store buffers are used to model write-read reordering on TSO, which account for the weak behaviour in Fig. 1a. Specifically, executing a write on TSO comprises two steps: (i) when a thread issues a write, the write is only recorded in its store buffer; (ii) writes in the buffer are debuffered (in FIFO order) and propagated to the local memory at a later point. When a thread issues a read from a location x, it first consults its own store buffer. If it

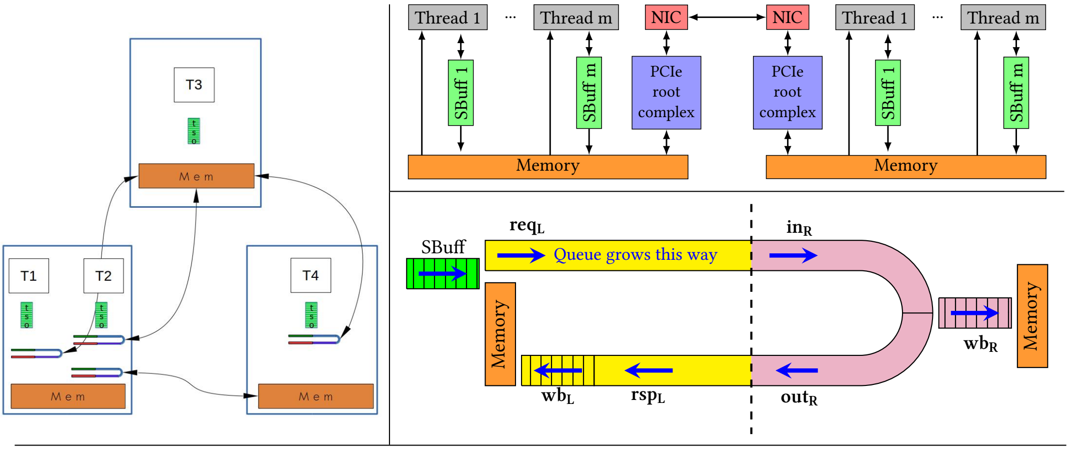

Fig. 4. RDMA$^{\text{TSO}}$ architecture overview (top, right); a possible RDMA network configuration with three nodes and four thread (left); the queue-pair structure (below, right)

contains a write for x, the thread reads the value of the latest such write; otherwise, the thread reads the value of x from its local memory. In other words, one can model the reordering of a write w after a later read r by *delaying* the debuffering of w until after r has executed. Moreover, executing an `mfence` or a CAS debuffers all its delayed writes in the store buffer and propagates them to memory (in FIFO order), thus preventing write-read reordering. That is, the only *CPU* operations that a store buffer contains are delayed writes. We describe the execution of polls shortly.

Remote Operations and Queue Pairs. To model the communication between the nodes, each thread t has a distinct *queue pair* for each remote node whose memory t accesses. For instance, the network configuration in the left of Fig. 4 comprises three nodes with four threads and with the queue pairs depicted as 'horse-shoes', where e.g. thread T2 in the bottom left node accesses the memories of the other two nodes, and it does so through two distinct queue pairs shown underneath T2. We refer to the queue pair of a thread towards node $\overline{n}$ as its $\overline{n}$-queue pair.

The details of a queue-pair structure is shown at the bottom-right of Fig. 4, where each queue pair of thread t is connected to the store buffer of t. As shown, an $\overline{n}$-queue pair comprises six FIFO buffers—$\mathbf{req}_\mathrm{L}$, $\mathbf{in}_\mathrm{R}$, $\mathbf{wb}_\mathrm{R}$, $\mathbf{out}_\mathrm{R}$, $\mathbf{rsp}_\mathrm{L}$, and $\mathbf{wb}_\mathrm{L}$—that we describe below. A queue pair contains *pending* operations if any of its $\mathbf{in}_\mathrm{R}$, $\mathbf{out}_\mathrm{R}$ or $\mathbf{rsp}_\mathrm{L}$ components is non-empty. The types of the entries in each of the six buffers of a queue pair is also summarised in Fig. 5.

Recall that a remote operation rc^n may either be a get, put or remote fence instruction. When t executes a remote operation rc^n towards node $\overline{n}$, it adds rc^n to its store buffer. That is, as well as CPU (delayed) writes, a store buffer may contain remote (get, put and fence) operations. Once rc^n is debuffered from the store buffer, it is forwarded to its $\overline{n}$-queue pair, where it travels through the queue-pair pipeline and is processed differently depending on the type of rc^n, as we describe below.

Request Buffers ($\mathbf{req}_\mathrm{L}$). The $\mathbf{req}_\mathrm{L}$ is the entry point of the queue pair, containing remote requests that are to be forwarded. It comprises a sequence of remote get, put and fence operations to be handled by the local NIC. When an entry rc^n reaches the head of $\mathbf{req}_\mathrm{L}$, it is processed as follows. (1) If rc^n is a get, then it is simply forwarded to the remote inbox $\mathbf{in}_\mathrm{R}$. (2) If rc^n is a put $x^{\overline{n}} := y$, then the value v of y is looked up in the local memory and $x^{\overline{n}} := v$ is forwarded to $\mathbf{in}_\mathrm{R}$; i.e. y in $x^{\overline{n}} := y$ is replaced with its current in-memory value. (3) If rc^n is a remote fence, then the execution on the queue pair is blocked until it has no pending operations, i.e. $\mathbf{in}_\mathrm{R}$, $\mathbf{out}_\mathrm{R}$ and $\mathbf{rsp}_\mathrm{L}$ are all empty.

	$y^{\overline{n}} := x^n$	$y^{\overline{n}} := v$	$\mathsf{ack_p}$	$x^n := y^{\overline{n}}$	$x^n := v$	cn	$\mathsf{rfence}\ \overline{n}$
b ∈ SBuff	✓			✓	✓		✓
$\mathbf{req_L}$	✓			✓			✓
$\mathbf{in_R}$		✓		✓			
$\mathbf{wb_R}$		✓					
$\mathbf{out_R}$			✓	✓	✓		
$\mathbf{rsp_L}$			✓		✓		
$\mathbf{wb_L}$					✓	✓	

Fig. 5. The types of entries in the store buffers (b) of a thread on n and its six buffers on the $\overline{n}$-queue pair

Inbox Buffers ($\mathbf{in_R}$). The $\mathbf{in_R}$ contains requests forwarded by $\mathbf{req_L}$ that are to be processed by $\overline{n}$, i.e. each $\mathbf{req_L}$ entry is either a get or a put with a value (of the form $y^{\overline{n}} := v$). Processing a put $y^{\overline{n}} := v$ (once at $\mathbf{in_R}$ head) forwards it to the remote write-back buffer $\mathbf{wb_R}$ and also sends a *put acknowledgement*, $\mathsf{ack_p}$, to $\mathbf{out_R}$. Processing a get $x := y^{\overline{n}}$ (once at $\mathbf{in_R}$ head) does not immediately fetch the y value from the $\overline{n}$ memory; rather, it forwards it to $\mathbf{out_R}$ to be fulfilled later in $\mathbf{out_R}$.

Outbox Buffers ($\mathbf{out_R}$). The $\mathbf{out_R}$ buffer contains requests processed or forwarded by $\mathbf{in_R}$, and thus each entry in $\mathbf{out_R}$ is either a put acknowledgement ($\mathsf{ack_p}$), a get operation $x := y^{\overline{n}}$ *yet to be fulfilled*, or a *fulfilled* get $x := v$. An acknowledgement or a fulfilled get is processed when it reaches the head of $\mathbf{out_R}$, whereupon it is simply forwarded to $\mathbf{rsp_L}$. By contrast, a yet-to-be-fulfilled get $x := y^{\overline{n}}$ may be fulfilled at *any time* (before reaching the $\mathbf{out_R}$ head), where the value v of y is fetched from the memory of $\overline{n}$, and $x := y^{\overline{n}}$ is transformed to the fulfilled get $x := v$ and left in $\mathbf{out_R}$. This fulfilled get is later processed once it reaches the head of $\mathbf{out_R}$, as described above.

Remote Write-Back Buffers ($\mathbf{wb_R}$). The $\mathbf{wb_R}$ buffer contains requests forwarded by $\mathbf{in_R}$, and thus each entry in $\mathbf{wb_R}$ is a put operation with a value (of the form $y^{\overline{n}} := v$). Processing $y^{\overline{n}} := v$ (at the head of $\mathbf{wb_R}$) simply removes it from $\mathbf{wb_R}$ and writes v to y in the memory of $\overline{n}$.

Response Buffers ($\mathbf{rsp_L}$). The $\mathbf{rsp_L}$ buffer contains processed requests forwarded by $\mathbf{out_R}$, and thus each entry in $\mathbf{rsp_L}$ is either an $\mathsf{ack_p}$, acknowledging a processed put, or a fulfilled get. Processing each $\mathbf{rsp_L}$ entry (once at the head of $\mathbf{rsp_L}$) generates a *completion notification*, cn, which is used to serve poll requests, as we describe shortly. Specifically, processing $\mathsf{ack_p}$ simply removes it from $\mathbf{rsp_L}$ and forwards a completion notification to $\mathbf{wb_L}$. Analogously, processing a fulfilled get $x := v$ simply forwards $x := v$ together with a completion notification to $\mathbf{wb_L}$.

Local Write-Back Buffers ($\mathbf{wb_L}$). The $\mathbf{wb_L}$ buffer contains processed requests forwarded by $\mathbf{rsp_L}$; i.e. each entry in $\mathbf{wb_L}$ is either a completion notification (cn, associated with processed gets and puts), or a fulfilled get (of the form $x := v$). Processing a fulfilled get $x := v$ simply removes it and writes v to x in the local memory. A completion notification is left in the $\mathbf{wb_L}$ and is only removed when the associated get/put operation is polled, as we describe below.[4]

Poll Operations. Once a get enters $\mathbf{req_L}$, it progresses through the pipeline as follows. (G_1) it is forwarded to $\mathbf{in_R}$; (G_2) it is forwarded to $\mathbf{out_R}$ without being fulfilled; (G_3) it is fulfilled at some point while in $\mathbf{out_R}$; (G_4) it is forwarded to $\mathbf{rsp_L}$; (G_5) it is forwarded to $\mathbf{wb_L}$ together with a completion notification (cn); and (G_6) it is removed from $\mathbf{wb_L}$ and its effect is written to memory.

Similarly, once a put operation $x^{\overline{n}} := y$ enters the queue-pair pipeline it proceeds as follows. (P_1) it is simplified to $x^{\overline{n}} := v$ (where v is the value of y in the local memory) and forwarded to $\mathbf{in_R}$; (P_2) it is forwarded to $\mathbf{wb_R}$ and simultaneously an acknowledgement $\mathsf{ack_p}$ is forwarded to $\mathbf{out_R}$;

[4] In some implementations, write-back buffers ($\mathbf{wb_L}$ and $\mathbf{wb_R}$) of different queue pairs may physically use the same hardware buffers in the PCIe fabric. This does not introduce any additional weak behaviours.

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

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

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

$$\frac{\mathsf{C} \rightsquigarrow \mathsf{C}'}{\mathsf{C} \xrightarrow{\varepsilon} \mathsf{C}'} \qquad \frac{\mathsf{elocs}(e) = \emptyset}{x := e \xrightarrow{\mathtt{lW}(x, [\![e]\!])} \mathtt{skip}} \qquad \frac{\mathsf{elocs}(e_{\text{old}}) = \mathsf{elocs}(e_{\text{new}}) = \emptyset \qquad v \neq [\![e_{\text{old}}]\!]}{z := \mathtt{CAS}(x, e_{\text{old}}, e_{\text{new}}) \xrightarrow{\mathtt{CASF}(x,v)} z := v}$$

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

$$\frac{}{\overline{y} := x \xrightarrow{\mathtt{Put}(\overline{y}, x)} \mathtt{skip}} \qquad \frac{}{\mathtt{rfence}\ \overline{n} \xrightarrow{\mathtt{rF}(\overline{n})} \mathtt{skip}} \qquad \frac{}{\mathtt{poll}(\overline{n}) \xrightarrow{\mathtt{P}(\overline{n})} \mathtt{skip}}$$

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

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

Fig. 6. The RDMA$^{\text{TSO}}$ program and command transitions

(P$_3$) its $\mathbf{wb_R}$ entry is eventually removed and applied to the remote memory; its associated $\mathsf{ack_p}$ in $\mathbf{out_R}$ (P$_4$) is forwarded to $\mathbf{rsp_L}$; and (P$_5$) later forwarded to $\mathbf{wb_L}$ as cn.

That is, both get and put operations result in a completion notification (cn) when complete. Indeed, this is precisely why we record an acknowledgement for each put: the $\mathsf{ack_p}$ serves as a placeholder for a processed put, so that we can generate an associated notification when complete.

Recall that $\mathtt{poll}(\overline{n})$ awaits the completion of the earliest unpolled get/put towards $\overline{n}$. To achieve this, completion notifications are left in $\mathbf{wb_L}$ until polled. Executing $\mathtt{poll}(\overline{n})$ can proceed if the head of $\mathbf{wb_L}$ of the $\overline{n}$-queue pair is a cn entry, in which case this cn entry (the earliest in FIFO order) is removed. If $\overline{n}$-$\mathbf{wb_L}$ is empty or its head is a write entry, then the execution of $\mathtt{poll}(\overline{n})$ is blocked.

3.2 RDMA$^{\text{TSO}}$ Concrete Operational Semantics

We describe the RDMA$^{\text{TSO}}$ concrete operational semantics by separating the transitions of its program and hardware subsystems. The former describe the steps in program execution, e.g. how a branching program is reduced. The latter describe how the memory, store buffers and queue pairs evolve throughout the execution, e.g. how remote puts reach the memory. The RDMA$^{\text{TSO}}$ operational semantics is then defined by combining the transitions of its program and hardware subsystems.

Program Transitions. Program transitions in the middle of Fig. 6 are defined via the transitions of their constituent commands. Command transitions are of the form $\mathsf{C} \xrightarrow{l} \mathsf{C}'$, where $\mathsf{C}, \mathsf{C}' \in \mathsf{Comm}$ are sequential commands and l is a *label*. A label is either ε for silent transitions, or in Lab (defined at the top of Fig. 6) for executing a primitive command. Labels are defined as the union of Lab_n for all nodes n. A label in Lab_n may be (1) $\mathtt{lW}(x^n, v)$, for a CPU write on location x with value v;

(2) $\mathtt{lR}(x^n, v)$, for a CPU read on x reading v; (3) $\mathtt{CASS}(x^n, v_1, v_2)$, for a successful CAS reading the expected value v_1 from x and updating it to v_2; (4) $\mathtt{CASF}(x^n, v)$, for a failed CAS where the expected value does not match the value v of x in memory; (5) F, for a memory fence; (6) $\mathsf{P}(\overline{n})$, for a poll on $\overline{n}$; (7) $\mathtt{Get}(x^n, y^{\overline{n}})$, for $x^n := y^{\overline{n}}$; (8) $\mathtt{Put}(y^{\overline{n}}, x^n)$, for $y^{\overline{n}} := x^n$; or (9) $\mathsf{rF}(\overline{n})$, for a remote fence on $\overline{n}$.

Command transitions for sequential composition, choice and loops (the top line) are standard. The next four transitions reduce assignments and CAS operations. As assignments and CAS involve expressions, their transitions *rewrite* expressions step-by-step, as defined at the bottom of Fig. 6 (via $\rightsquigarrow$ transitions). Each rewrite step of assignment rewrites $x := e$ as $\mathtt{assume}(y = v); x := e[v/y]$, replacing a location y in e with an arbitrary value v and checking that v matches the value of y via $\mathtt{assume}(y = v)$. Note that value v is unconstrained at this point and is later constrained when connecting program transitions with memory ones. The two rewrite transitions for CAS are analogous. Observe that when $\mathsf{C} \rightsquigarrow \mathsf{C}'$, then C silently transitions to C' (with label ε). Once e in $x := e$ is closed (i.e. contains no locations), then it is reduced to $\mathtt{skip}$ with the corresponding CPU write label for writing the value of $[[e]]$ to x; *mutatis mutandis* for CAS operations. The other transitions reduce memory fences, gets, puts, remote fences, polls and assumes to $\mathtt{skip}$ with matching labels. Finally, $\mathtt{assume}(x{=}v)$ (resp. $\mathtt{assume}(x{\neq}v')$) reduces to $\mathtt{skip}$ with a $\mathtt{lR}(x, v)$ label when the in-memory value v of x matches (resp. does not match) the assertion.

Program transitions are of the form $\mathsf{P} \xrightarrow{t:l} \mathsf{P}'$, where P, P' denote (concurrent) programs, t is the identifier of the executing thread, and l is the transition label. Program transitions simply lift the transitions of their constituent threads (the bottom right rule).

Hardware Transitions. The RDMA$^{\text{TSO}}$ *hardware transitions*, given in the middle of Fig. 7, are of the form $\mathsf{M}, \mathsf{B}, \mathsf{QP} \xrightarrow{t:l} \mathsf{M}', \mathsf{B}', \mathsf{QP}'$, where M, M' are global *memories*, B, B' are *store-buffer maps* and $\mathsf{QP}, \mathsf{QP}'$ are *queue-pair maps*, defined at the top of Fig. 7. We model a memory as a map from locations to values. A store-buffer map is a function mapping each thread t to a *store buffer* on its associated node $n(t)$. A store buffer b on node n is a sequence of CPU writes and RDMA puts, gets and fences, as described in §3.1 (see Fig. 5). A queue-pair map is a function mapping each thread t (on node $n(t)$) to another function associating each non-$n(t)$ node with a queue pair. That is, each thread t is associated with a set of queue pairs, one for each other node on the network. A queue pair qp is a tuple of six buffers, $\mathbf{req_L}$, $\mathbf{in_R}$, $\mathbf{wb_R}$, $\mathbf{out_R}$, $\mathbf{rsp_L}$ and $\mathbf{wb_L}$, as described in §3.1. Each queue-pair buffer in turn is a sequence of entries as prescribed in Fig. 5. We write ‘qp.’ to project components of qp and use a standard map update notation to modify these components.

The first five hardware transitions describe the execution of CPU operations, as described in §3.1. Specifically, when a thread writes v to x, it records this write in its store buffer (first transition). Recall that when a thread t reads from x, it first consults its own store buffer, followed by the memory if no write to x is found in the store buffer. This lookup chain is captured by $\mathsf{M} \lhd \mathsf{B}(t)$ (second transition), defined below the hardware transitions in Fig. 7. The execution of a CAS or $\mathtt{mfence}$ proceeds if the store buffer of the executing thread is empty (the next three transitions).

The next transition describes the debuffering of a CPU write and propagating it to the memory. Similarly, the transition after describes debuffering a remote operation rc^n towards node $\overline{n}$, where it is removed from the store buffer and appended to the $\mathbf{req_L}$ component of the $\overline{n}$-queue pair.

The next three transitions describe executing a remote get, put or fence operation, where a corresponding entry is added to the store buffer. The penultimate transition describes executing a poll on $\overline{n}$, which removes the earliest completion notification in $\mathbf{wb_L}$ of the $\overline{n}$-queue pair. The last transition describes how the queue-pair entries travel through its six buffers, captured by the *queue-pair transitions* ($\rightarrow_{\mathsf{qp}}$) defined at the bottom of Fig. 7 (ignoring highlights for now).

$$\mathsf{M} \in \mathsf{Mem} \triangleq \mathsf{Loc} \to \mathsf{Val} \qquad \mathsf{B} \in \mathsf{SBMap} \triangleq \lambda t \in \mathsf{Tid}.\mathsf{SBuff}_{n(t)} \qquad \mathsf{QP} \in \mathsf{QPMap} \triangleq \lambda t.\left(\lambda \overline{n(t)}.\mathsf{QPair}^{\overline{n}}_{n}\right)$$

$$\mathsf{b} \in \mathsf{SBuff}_n \triangleq \left\{x^n{:=}v, y^{\overline{n}}{:=}x^n, x^n{:=}y^{\overline{n}}, \mathtt{rfence}\ \overline{n}\right\}^* \qquad \mathsf{qp} \in \mathsf{QPair}^{\overline{n}}_n \triangleq \mathsf{Req}^{\overline{n}}_n \times \mathsf{In}^{\overline{n}}_n \times \mathsf{WBR}^{\overline{n}}_n \times \mathsf{Out}^{\overline{n}}_n \times \mathsf{Rsp}^{\overline{n}}_n \times \mathsf{WBL}^{\overline{n}}_n$$

$$\mathbf{req_L} \in \mathsf{Req}^{\overline{n}}_n \triangleq \left\{y^{\overline{n}}{:=}x^n, x^n{:=}y^{\overline{n}}, \mathtt{rfence}\ \overline{n}\right\}^* \quad \mathbf{in_R} \in \mathsf{In}^{\overline{n}}_n \triangleq \left\{y^{\overline{n}} := v, x^n := y^{\overline{n}}\right\}^* \quad \mathbf{wb_L} \in \mathsf{WBL}^{\overline{n}}_n \triangleq \left\{\mathsf{cn}, x^n := v\right\}^*$$

$$\mathbf{out_R} \in \mathsf{Out}^{\overline{n}}_n \triangleq \left\{\mathsf{ack_p}, x^n := v, x^n := y^{\overline{n}}\right\}^* \qquad \mathbf{rsp_L} \in \mathsf{Rsp}^{\overline{n}}_n \triangleq \left\{\mathsf{ack_p}, x^n := v\right\}^* \qquad \mathbf{wb_R} \in \mathsf{WBR}^{\overline{n}}_n \triangleq \left\{y^{\overline{n}} := v\right\}^*$$

$$\dfrac{\mathsf{B}' = \mathsf{B}[t \mapsto (x := v) \cdot \mathsf{B}(t)]}{\mathsf{M}, \mathsf{B}, \mathsf{QP} \xrightarrow{t:\mathtt{lW}(x,v)} \mathsf{M}, \mathsf{B}', \mathsf{QP}} \qquad \dfrac{(\mathsf{M} \lhd \mathsf{B}(t))(x) = v}{\mathsf{M}, \mathsf{B}, \mathsf{QP} \xrightarrow{t:\mathtt{lR}(x,v)} \mathsf{M}, \mathsf{B}, \mathsf{QP}} \qquad \dfrac{\mathsf{B}(t) = \varepsilon \qquad \mathsf{M}(x) = v_1}{\mathsf{M}, \mathsf{B}, \mathsf{QP} \xrightarrow{t:\mathsf{CASS}(x,v_1,v_2)} \mathsf{M}[x \mapsto v_2], \mathsf{B}, \mathsf{QP}}$$

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

$$\dfrac{\mathsf{B}(t) = \mathsf{b} \cdot \mathsf{rc}^n \qquad \mathsf{rc}^n \in \left\{x := y^{\overline{n}}, y^{\overline{n}} := x, \mathtt{rfence}\ \overline{n}\right\} \qquad \mathsf{QP}(t)(\overline{n}) = \mathsf{qp} \qquad \mathsf{qp}' = \mathsf{qp}[\mathbf{req_L} \mapsto \mathsf{rc}^n \cdot \mathsf{qp}.\mathbf{req_L}]}{\mathsf{M}, \mathsf{B}, \mathsf{QP} \xrightarrow{t:\varepsilon} \mathsf{M}, \mathsf{B}[t \mapsto \mathsf{b}], \mathsf{QP}[t \mapsto \mathsf{QP}(t)[\overline{n} \mapsto \mathsf{qp}']]}$$

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

$$\dfrac{\mathsf{QP}(t)(\overline{n}) = \mathsf{qp} \qquad \mathsf{qp}.\mathbf{wb_L} = \alpha \cdot \mathsf{cn} \qquad \mathsf{qp}' = \mathsf{qp}[\mathbf{wb_L} \mapsto \alpha]}{\mathsf{M}, \mathsf{B}, \mathsf{QP} \xrightarrow{t:\mathsf{P}(\overline{n})} \mathsf{M}, \mathsf{B}, \mathsf{QP}[t \mapsto \mathsf{QP}(t)[\overline{n} \mapsto \mathsf{qp}']]} \qquad \dfrac{\mathsf{M}, \mathsf{QP}(t)(\overline{n}) \to_{\mathsf{qp}} \mathsf{M}', \mathsf{qp}}{\mathsf{M}, \mathsf{B}, \mathsf{QP} \xrightarrow{t:\varepsilon} \mathsf{M}', \mathsf{B}, \mathsf{QP}[t \mapsto \mathsf{QP}(t)[\overline{n} \mapsto \mathsf{qp}]]}$$

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

$$\dfrac{\begin{array}{llll} & \mathsf{qp}.\mathbf{req_L} = \alpha \cdot (x{:=}\overline{y}) & \mathsf{qp}' = \mathsf{qp}[\mathbf{req_L} \mapsto \alpha][\mathbf{in_R} \mapsto (x{:=}\overline{y}) \cdot \mathsf{qp}.\mathbf{in_R}] & \mathsf{M}' = \mathsf{M} \\ \vee & \mathsf{qp}.\mathbf{in_R} = \alpha \cdot (x{:=}\overline{y}) & \mathsf{qp}' = \mathsf{qp}[\mathbf{in_R} \mapsto \alpha][\mathbf{out_R} \mapsto (x{:=}\overline{y}) \cdot \mathsf{qp}.\mathbf{out_R}] & \mathsf{M}' = \mathsf{M} \\ \vee & \mathsf{qp}.\mathbf{out_R} = \alpha \cdot (x{:=}\overline{y}) \cdot \beta & \mathsf{qp}.\mathbf{wb_R} = \varepsilon \qquad \mathsf{qp}' = \mathsf{qp}[\mathbf{out_R} \mapsto \alpha \cdot (x{:=}\mathsf{M}(\overline{y})) \cdot \beta] & \mathsf{M}' = \mathsf{M} \\ \vee & \mathsf{qp}.\mathbf{out_R} = \alpha \cdot (x{:=}v) & \mathsf{qp}' = \mathsf{qp}[\mathbf{out_R} \mapsto \alpha][\mathbf{rsp_L} \mapsto (x{:=}v) \cdot \mathsf{qp}.\mathbf{rsp_L}] & \mathsf{M}' = \mathsf{M} \\ \vee & \mathsf{qp}.\mathbf{rsp_L} = \alpha \cdot (x{:=}v) & \mathsf{qp}' = \mathsf{qp}[\mathbf{rsp_L} \mapsto \alpha][\mathbf{wb_L} \mapsto \mathsf{cn} \cdot (x{:=}v) \cdot \mathsf{qp}.\mathbf{wb_L}] & \mathsf{M}' = \mathsf{M} \\ \vee & \mathsf{qp}.\mathbf{wb_L} = \alpha \cdot (x{:=}v) \cdot \mathsf{cn}^* & \mathsf{qp}' = \mathsf{qp}[\mathbf{wb_L} \mapsto \alpha \cdot \mathsf{cn}^*] & \mathsf{M}' = \mathsf{M}[x \mapsto v] \end{array}}{\mathsf{M}, \mathsf{qp} \to_{\mathsf{qp}} \mathsf{M}', \mathsf{qp}'}$$

$$\dfrac{\begin{array}{llll} & \mathsf{qp}.\mathbf{req_L} = \alpha \cdot (\overline{y}{:=}x) & \mathsf{qp}.\mathbf{wb_L} = \mathsf{cn}^* \qquad \mathsf{qp}' = \mathsf{qp}[\mathbf{req_L} \mapsto \alpha][\mathbf{in_R} \mapsto (\overline{y}{:=}\ \mathsf{M}(x)) \cdot \mathsf{qp}.\mathbf{in_R}] & \mathsf{M}' = \mathsf{M} \\ \vee & \mathsf{qp}.\mathbf{in_R} = \alpha \cdot (\overline{y}{:=}v) & \mathsf{qp}' = \mathsf{qp}[\mathbf{in_R} \mapsto \alpha][\mathbf{wb_R} \mapsto (\overline{y}{:=}v) \cdot \mathsf{qp}.\mathbf{wb_R}][\mathbf{out_R} \mapsto \mathsf{ack_p} \cdot \mathsf{qp}.\mathbf{out_R}] & \mathsf{M}' = \mathsf{M} \\ \vee & \mathsf{qp}.\mathbf{wb_R} = \alpha \cdot (\overline{y}{:=}v) & \mathsf{qp}' = \mathsf{qp}[\mathbf{wb_R} \mapsto \alpha] & \mathsf{M}' = \mathsf{M}[\overline{y} \mapsto v] \\ \vee & \mathsf{qp}.\mathbf{out_R} = \alpha \cdot \mathsf{ack_p} & \mathsf{qp}' = \mathsf{qp}[\mathbf{out_R} \mapsto \alpha][\mathbf{rsp_L} \mapsto \mathsf{ack_p} \cdot \mathsf{qp}.\mathbf{rsp_L}] & \mathsf{M}' = \mathsf{M} \\ \vee & \mathsf{qp}.\mathbf{rsp_L} = \alpha \cdot \mathsf{ack_p} & \mathsf{qp}' = \mathsf{qp}[\mathbf{rsp_L} \mapsto \alpha][\mathbf{wb_L} \mapsto \mathsf{cn} \cdot \mathsf{qp}.\mathbf{wb_L}] & \mathsf{M}' = \mathsf{M} \end{array}}{\mathsf{M}, \mathsf{qp} \to_{\mathsf{qp}} \mathsf{M}', \mathsf{qp}'}$$

$$\dfrac{\mathsf{qp}.\mathbf{req_L} = \alpha \cdot (\mathtt{rfence}\ \overline{n}) \qquad \mathsf{qp}.\mathbf{in_R} = \mathsf{qp}.\mathbf{out_R} = \mathsf{qp}.\mathbf{rsp_L} = \varepsilon \qquad \mathsf{qp}' = \mathsf{qp}[\mathbf{req_L} \mapsto \alpha]}{\mathsf{M}, \mathsf{qp} \to_{\mathsf{qp}} \mathsf{M}, \mathsf{qp}'}$$

Fig. 7. RDMA$^{\mathsf{TSO}}$ hardware domains (above), hardware transitions (middle) and queue-pair transitions (below)

Recall from §3.1 that once a get operation enters a queue pair it travels through the queue-pair pipeline in six steps, i.e. G_1–G_6 on p. 11. This is captured by the top $\to_{\mathsf{qp}}$ transition at the bottom of Fig. 7, where for brevity we have combined these six steps into one transition with a disjunctive premise, with each disjunct corresponding to a step in G_1–G_6. That is, if any of the six disjuncts in

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

Fig. 8. RDMA$^{\text{TSO}}$ operational semantics with the program and hardware transitions given in Fig. 6 and Fig. 7

the premise of the rule hold, then the transition is enabled. Analogously, the second $\rightarrow_{\mathsf{qp}}$ transition describes how a put operation proceeds through the queue-pair pipeline, with each of the five disjuncts describing one of the five steps in P_1–P_5 (p. 11). Finally, the last $\rightarrow_{\mathsf{qp}}$ transition describes the execution of a remote fence as described in §3.1, ensuring that it can only proceed if there are no pending operations on the queue pair (i.e. $\mathsf{qp}.\mathbf{in_R} = \mathsf{qp}.\mathbf{out_R} = \mathsf{qp}.\mathbf{rsp_L} = \varepsilon$).

RDMA$^{\text{TSO}}$ Combined Transitions. The RDMA$^{\text{TSO}}$ operational semantics is defined by combining its program and hardware transitions as shown in Fig. 8. When the program subsystem takes a silent step, then the hardware subsystem is unchanged (first transition); analogously, when the hardware subsystem takes a silent step, then the program subsystem is unchanged (second transition). Finally, when the program and hardware subsystems both take the same transition (with the same label and by the same thread), then the transition effect is that of their combined effects.

Operational Semantics without PCIe. Recall from §2 (p. 8) that we model the PCIe-specific guarantee where a NIC remote read propagates all pending NIC remote writes (in $\mathbf{wb_R}$) to memory, while a NIC local read propagates all pending NIC local writes (in $\mathbf{wb_L}$) to memory. Nevertheless, we can relax this as follows. For NIC remote reads, we can replace the highlighted premises of the get queue-pair transition (top $\rightarrow_{\mathsf{qp}}$ transition) in Fig. 7 with $\mathsf{qp}' = \mathsf{qp}[\mathbf{out_R} \mapsto \alpha \cdot (x{:=}(\mathsf{M} \triangleleft \mathbf{wb_R})(\overline{y})) \cdot \beta]$. That is, we no longer require $\mathbf{wb_R}$ to be empty (i.e. there may be pending writes in $\mathbf{wb_R}$), and when reading the value of $\overline{y}$ we first check for pending writes on $\overline{y}$ in $\mathbf{wb_R}$.

For NIC local reads, we can similarly replace the highlighted premises of the put queue-pair transition (middle $\rightarrow_{\mathsf{qp}}$ transition) with $\mathsf{qp}' = \mathsf{qp}[\mathbf{req_L} \mapsto \alpha][\mathbf{in_R} \mapsto (\overline{y} := (\mathsf{M} \triangleleft \mathbf{wb_L})(x)) \cdot \mathsf{qp}.\mathbf{in_R}]$. That is, we no longer require $\mathbf{wb_L}$ to contain only completion notifications (and allow it also to contain pending writes), and we first check $\mathbf{wb_L}$ for pending writes when reading the value of x.

Observations. Given a thread t and its store buffer b, all remote operations by t also go through b. Hence, as store buffers are FIFO, a CPU write cc in b before a remote operation rc in b always reaches the memory before rc is debuffered, and thus cc is visible to rc. Moreover, as all six buffers of the queue-pair pipeline are FIFO, remote operations maintain the order in which they were issued as they go through the queue-pair pipeline. Therefore, a thread always receives the completion notifications of get/put operations in the order they were submitted (i.e. the program order).

Observe that an `rFence` stipulates that only $\mathbf{in_R}$, $\mathbf{out_R}$ and $\mathbf{rsp_L}$ be empty but not $\mathbf{wb_L}$ and $\mathbf{wb_R}$. As such, `rFence` cannot guarantee that the result of earlier put (resp. get) operations have reached the remote (resp. local) memory: they can still be pending in $\mathbf{wb_R}$ (resp. $\mathbf{wb_L}$). Moreover, `rFence` has no bearing on CPU operations and does not block their execution. Hence, later CPU operations (after `rFence`) may be visible to earlier get/put operations (those before `rFence`).

Recall that a put operation $x^n := y$ comprises a local read from y and a remote write to x, and a get operation $x := y^n$ comprises a remote read from y and a local write to x. Note that the local read of a put is fulfilled when it reaches the head of $\mathbf{in_R}$ and is subsequently forwarded to $\mathbf{out_R}$. This ensures that puts are executed in program order and that puts are executed before all later gets (as in Fig. 2g). By contrast, the remote read of a get is fulfilled while in $\mathbf{out_R}$ *non-deterministically* (i.e. not necessarily when it is at the head of $\mathbf{out_R}$). This means that remote reads of gets can be *reordered* with respect to one another, as well as with respect to the remote writes of puts (as in Fig. 2h). Such reorderings can be prevented by adding an `rFence` after a get (as in Fig. 2i).

Lastly, note that a `poll` retrieves the earliest `cn` in $\mathbf{wb_L}$ (i.e. at its head). In the case of gets, the result of the get (its local write) is sent to $\mathbf{wb_L}$ before its associated `cn`. As such, if the head of $\mathbf{wb_L}$ is `cn`, then its result is guaranteed to have reached the memory of the local node when polling. By contrast, in the case of puts, its remote write operation could still be in $\mathbf{wb_R}$ when polling, and thus polling a put cannot guarantee that the effect of the put has reached the remote memory.

3.3 RDMA^TSO Simplified Operational Semantics

The concrete operational semantics in §3.2 reflects the structure of the underlying hardware, namely that of the six buffers in a queue pair. However, since all four buffers ($\mathbf{req_L}$, $\mathbf{in_R}$, $\mathbf{out_R}$, $\mathbf{rsp_L}$) in the middle are FIFO, we can simplify them by modelling them as a *single* buffer. Specifically, we model a *simple queue pair*, $\mathbf{sqp} \in \mathsf{SQPair}_n^{\overline{n}} \triangleq \mathsf{Pipe}_n^{\overline{n}} \times \mathsf{WBR}_n^{\overline{n}} \times \mathsf{WBL}_n^{\overline{n}}$, as a tuple comprising a *pipe*, as well as local and remote write-back buffers (as before). A pipe, $\mathbf{pipe} \in \mathsf{Pipe}_n^{\overline{n}} \triangleq \left\{y^{\overline{n}} := x^n, y^{\overline{n}} := v, \mathsf{ack_p}, x^n := y^{\overline{n}}, x^n := v, \texttt{rfence}\ \overline{n}\right\}^*$, is simply a sequence of puts, gets, simplified puts and gets (with values), acknowledgements, and remote fences.

The program and command transitions of our simplified semantics is identical to those of the concrete one (Fig. 6). Moreover, the hardware transitions of our simplified semantics are those of the concrete semantics (Fig. 7), except that the queue-pair transitions at the bottom of Fig. 7 are replaced with the *simplified queue-pair transitions* available in the extended version. As before, the simplified operational semantics is obtained by combining its program and hardware transitions (Fig. 8).

Finally, we show that our concrete operational semantics is equivalent to the simplified one. We have mechanised our proof in Coq, available in the supplementary material [Ambal et al. 2024b], with an overview of the proof available in the extended version.

Theorem 3.1. *The concrete* RDMA^TSO *operational semantics is equivalent to the simplified one.*

4 RDMA^TSO 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 4.1 (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:

- (CPU) local read: $l = \mathtt{lR}(x^n, v_\mathrm{r})$
- (CPU) local write: $l = \mathtt{lW}(x^n, v_\mathrm{w})$
- (CPU) CAS: $l = \mathtt{CAS}(x^n, v_\mathrm{r}, v_\mathrm{w})$
- (CPU) memory fence: $l = \mathtt{F}$
- (CPU) poll: $l = \mathtt{P}(\overline{n})$
- NIC local read: $l = \mathtt{nlR}(x^n, v_\mathrm{r}, \overline{n})$
- NIC remote write: $l = \mathtt{nrW}(y^{\overline{n}}, v_\mathrm{w})$
- NIC remote read: $l = \mathtt{nrR}(y^{\overline{n}}, v_\mathrm{r})$
- NIC local write: $l = \mathtt{nlW}(x^n, v_\mathrm{w}, \overline{n})$
- NIC fence: $l = \mathtt{nF}(\overline{n})$

Each event label denotes whether the associated primitive command is handled by the NIC (right column, prefixed with `n`), or the CPU (left column). A poll instruction is handled by the CPU (it simply awaits for a completion notification from the NIC). A put operation $x^n := y$, which consists of a NIC local read from y and a NIC remote write to x, is modelled as two events of type `nlR` (on y) and `nrW` (on x). Similarly, a get $x := y^n$ is modelled as two events of type `nrR` (on y) and `nlW` (on x).

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

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 the local or remote memory. As such, since writes (be they by the CPU or NIC) are the only instructions that have an observable effect on memory, the time points at which they are issued and observed may differ. A CPU write is issued when it is added to the store buffer and is observed when it is debuffered and propagated to memory. Similarly, the local (resp. remote) write of a get (resp. put) is issued when it is added to $\mathbf{wb_L}$ (resp. $\mathbf{wb_R}$), and observed when it is propagated to memory. By contrast, all other events are *instantaneous* in that *either* they do not have an observable effect on memory and thus their issue and observation points coincide, *or* their effect is written to memory *immediately*. In particular, CAS operations are instantaneous. Note that the observation point of any instruction either coincides with its issue point (instantaneous events) or it follows its issue point (write events).

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 the relational composition of r_1 and r_2: $\{(a, b) \mid \exists c.\, (a, c) \in r_1 \land (c, b) \in r_2\}$. When r is a strict partial order, we write $r|_{\mathrm{imm}}$ 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 $\{\mathrm{e} \in E \mid \mathtt{loc}(\mathrm{e}) = x\}$. Given a set of events E and a label type $\mathtt{X}$, we write $E.\mathtt{X}$ for $\{\mathrm{e} \in E \mid \mathtt{type}(\mathrm{e}) = \mathtt{X}\}$, 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 $E.\mathcal{W} \triangleq E.\mathtt{lW} \cup E.\mathtt{CAS} \cup E.\mathtt{nlW} \cup E.\mathtt{nrW}$, *NIC writes* as $E.\mathtt{nW} \triangleq E.\mathtt{nlW} \cup E.\mathtt{nrW}$ and *instantaneous events* as $E.\mathtt{Inst} \triangleq E \setminus (E.\mathtt{lW} \cup E.\mathtt{nlW} \cup E.\mathtt{nrW})$. Intuitively, the effects of CPU writes, NIC local writes and NIC remote writes (labelled $\mathtt{lW}$, $\mathtt{nlW}$ and $\mathtt{nrW}$) are only visible when they respectively leave the store buffer, $\mathbf{wb_L}$, and $\mathbf{wb_R}$, and are thus excluded from the set of instantaneous events.

The '*same-location*' relation is $\mathsf{sloc} \triangleq \left\{(\mathrm{e}, \mathrm{e}') \in \mathsf{Event}^2 \mid \mathtt{loc}(\mathrm{e}) = \mathtt{loc}(\mathrm{e}')\right\}$; the '*same-thread*' relation is $\mathsf{sthd} \triangleq \left\{(\mathrm{e}, \mathrm{e}') \in \mathsf{Event}^2 \mid t(\mathrm{e}) = t(\mathrm{e}')\right\}$; and the '*same-queue-pair*' relation is $\mathsf{sqp} \triangleq \{(\mathrm{e}, \mathrm{e}') \in \mathsf{Event}^2 \mid t(\mathrm{e}) = t(\mathrm{e}') \land \overline{n}(\mathrm{e}) = \overline{n}(\mathrm{e}')\}$. Note that $\mathsf{sqp} \subseteq \mathsf{sthd}$ and that sloc, sthd and 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 $E.\mathsf{sqp}$.

Definition 4.2 (Pre-executions). A *pre-execution* is a tuple $G = \langle E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{pf}, \mathsf{nfo} \rangle$, where:

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

Recall from §2 that we model the PCIe-specific guarantee where a NIC local read (`nlR`) propagates all pending NIC local writes (`nlW`) in $\mathbf{wb_L}$ (on the same queue pair) to memory, while a NIC remote read (`nrR`) propagates all pending NIC remote writes (`nrW`) in $\mathbf{wb_R}$ (on the same queue pair) to memory. In other words, the issue point of an `nlR` event e_r is totally ordered with respect to the issue and observation points of any `nlW` event e_w on the same queue pair. Specifically, either (1) e_w had already been flushed before issuing e_r (i.e. e_w had already left $\mathbf{wb_L}$) and thus e_w is issued and observed before e_r; or (2) e_w was in $\mathbf{wb_L}$ before issuing e_r, in which case we cannot issue e_r until $\mathbf{wb_L}$ is emptied, propagating e_w to memory, and thus e_w is issued and observed before e_r; or (3) e_w has not reached $\mathbf{wb_L}$ when we issue e_r, in which case e_w will reach and leave $\mathbf{wb_L}$ (i.e. is issued and observed) after e_r. Similarly for `nrR`/`nrW` events. 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. Later we describe how we can relax this PCIe guarantee (see p. 21).

Derived Relations. Given a pre-execution $\langle E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{pf}, \mathsf{nfo}\rangle$, 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. We further define the rf-*buffer* relation as $\mathsf{rf_b} \triangleq [\mathtt{lW}]; (\mathsf{rf} \cap \mathsf{sthd}); [\mathtt{lR}]$, including CPU rf edges by the same thread (with access to the same store buffer). We define the $\mathsf{rf_b}$-complement as $\mathsf{rf}_{\bar{\mathsf{b}}} \triangleq \mathsf{rf} \setminus \mathsf{rf_b}$, including all other rf edges (i.e. by different threads or involving remote operations). Intuitively, when $w \xrightarrow{\mathsf{rf_b}} r$, then r may read from w *before it is observable*. Specifically, as CPU writes are delayed in the store buffer and CPU reads first check the buffer, r can read from w either when (1) w is still in the thread's store buffer (i.e. w is not yet observable); or (2) w is in the memory (i.e. w is observable). By contrast, when $w \xrightarrow{\mathsf{rf}_{\bar{\mathsf{b}}}} r$, then r reads from w only once it is observable (i.e. it has reached the memory). Analogously, we define the rb-*buffer* relation as $\mathsf{rb_b} \triangleq [\mathtt{lR}]; (\mathsf{rb} \cap \mathsf{sthd}); [\mathtt{lW}]$.

Recall that we distinguish between the issue and observation points of an event. We thus define the '*issue-preserved program order*', ippo, as the subset of po edges ($\mathsf{ippo} \subseteq \mathsf{po}$) 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 reordered and thus issued in either order. The table at the top of Fig. 9 describes which po edges are included in ippo, where ✓ denotes that the two instructions are ippo-related (i.e. their issue order is preserved and they must be issued in program order), ✗ denotes that they are not ippo-related (i.e. their issue order is not preserved and 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 a NIC one, then they are issued in order (top-right quadrant of the table). By contrast, when a NIC instruction is followed by a CPU one, then they may be reordered (bottom-left quadrant), *as if* the NIC instruction was executed in a parallel thread (as discussed in §2), resulting in weak behaviours such as $z = 1$ in Fig. 2b.

Analogously, we define the '*observation-preserved program order*', oppo, as the subset of po edges ($\mathsf{oppo} \subseteq \mathsf{po}$) that must be preserved when observing the effects of instructions. In other words, if two events are oppo-related, then they become observable in program order; otherwise they may be reordered and become observable in either order. The table at the bottom of Fig. 9 describes which po edges are included in oppo, with ✓, ✗ and `sqp` interpreted in the same way as for ippo.

Observe that ippo and oppo only differ in four cells. In the case of B1 and B5, this is because CPU writes (type `lW`) are delayed in the store buffer and may be reordered and thus become observable after CPU operations that do not go through the store buffer, namely CPU reads and polls. Analogously, in the case of G10 (resp. I10), this is because NIC remote (resp. local) writes are delayed in $\mathbf{wb_R}$ (resp. $\mathbf{wb_L}$), so a later remote fence do not ensure the writes have propagated to memory.

Later in Program Order

Earlier in Program Order

ippo			CPU					NIC				
			1	2	3	4	5	6	7	8	9	10
			lR	lW	CAS	F	P	nlR	nrW	nrR	nlW	nF
CPU	A	lR	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	B	lW	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	C	CAS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	D	F	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	E	P	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NIC	F	nlR	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp
	G	nrW	✗	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp
	H	nrR	✗	✗	✗	✗	✗	✗	✗	✗	sqp	sqp
	I	nlW	✗	✗	✗	✗	✗	✗	✗	✗	sqp	sqp
	J	nF	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp

Later in Program Order

Earlier in Program Order

oppo			CPU					NIC				
			1	2	3	4	5	6	7	8	9	10
			lR	lW	CAS	F	P	nlR	nrW	nrR	nlW	nF
CPU	A	lR	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	B	lW	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓
	C	CAS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	D	F	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	E	P	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
NIC	F	nlR	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp
	G	nrW	✗	✗	✗	✗	✗	✗	sqp	sqp	sqp	✗
	H	nrR	✗	✗	✗	✗	✗	✗	✗	✗	sqp	sqp
	I	nlW	✗	✗	✗	✗	✗	✗	✗	✗	sqp	✗
	J	nF	✗	✗	✗	✗	✗	sqp	sqp	sqp	sqp	sqp

Fig. 9. The RDMA$^{\text{TSO}}$ ordering constraints on ippo (above) and oppo (below), where ✓ denotes that instructions are ordered (and cannot be reordered), ✗ denotes they are not ordered (and may be reordered), and sqp denotes they are ordered iff they are on the same queue pair. Replacing sqp in the highlighted cells (G8-G9) with ✗ would yield oppo for the weaker model without the PCIe guarantee of reads flushing buffers (p. 21).

From Programs to Executions. The *semantics* of a program P is a set of *consistent* executions (defined shortly) of P, defined by induction on the structure of P. This definition is standard and omitted (see the extended version). The executions produced by this construction are *well-formed*, as we define below.

Definition 4.3 (Executions). A pre-execution $G = \langle E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{pf}, \mathsf{nfo}\rangle$ is *well-formed* if the following hold for all w, r, w_1, w_2, p_2:

(1) Poll events poll-from the oldest non-polled remote operation on the same queue pair: if $w_1 \in G.\mathtt{nW}$ and $w_1 \xrightarrow{\mathsf{po}\cap\mathsf{sqp}} w_2 \xrightarrow{\mathsf{pf}} p_2$, then there exists p_1 such that $w_1 \xrightarrow{\mathsf{pf}} p_1 \xrightarrow{\mathsf{po}} p_2$.

(2) Each put (resp. get) operation corresponds to two events: a read and a write with the read immediately preceding the write in po: (a) if $r \in G.\mathtt{nlR}$ (resp. $r \in G.\mathtt{nrR}$), then $(r, w) \in \mathsf{po}|_{\mathrm{imm}}$ for some $w \in G.\mathtt{nrW}$ ($w \in G.\mathtt{nlW}$); and (b) if $w \in G.\mathtt{nrW}$ (resp. $w \in G.\mathtt{nlW}$), then $(r, w) \in \mathsf{po}|_{\mathrm{imm}}$ for some $r \in G.\mathtt{nlR}$ ($r \in G.\mathtt{nrR}$).

(3) Read and write events of a put (resp. get) have matching values: if $(r, w) \in G.\mathsf{po}|_{\mathrm{imm}}$, $\mathrm{type}(r) \in \{\mathtt{nlR}, \mathtt{nrR}\}$ and $\mathrm{type}(w) \in \{\mathtt{nlW}, \mathtt{nrW}\}$, then $v_{\mathrm{r}}(r) = v_{\mathrm{w}}(w)$.

An *execution* is a pre-execution (Def. 4.2) that is well-formed.

We use 'G.' to project the components and (derived) relations of execution G; e.g. G.rf and G.ippo. When the choice of G is clear, we simply write e.g. rf and ippo. See the extended version for execution examples.

Consistency. Note that the notion of an execution (Def. 4.3) imposes very few constraints on its po, rf, mo, pf 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 4.4 (RDMA$^{\text{TSO}}$-consistency). An execution $\langle E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{pf}, \mathsf{nfo}\rangle$ is *RDMA$^{\text{TSO}}$-consistent* iff (1) ib is irreflexive; (2) ob is irreflexive; and (3) $([\mathtt{Inst}]; \mathsf{ib}; \mathsf{ob})^+$ is irreflexive, where:

$$\mathsf{ib} \triangleq \left(\mathsf{ippo} \cup \mathsf{rf} \cup \mathsf{pf} \cup \mathsf{nfo} \cup \mathsf{rb}_{\mathsf{b}}\right)^+ \qquad \text{('issued-before')}$$

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

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$^{\text{TSO}}$-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 rb_{b} component in ib ensures that a CPU read r by thread t on location x observes all earlier writes on x by t: if r in t reads from w_r ($w_r \xrightarrow{\mathsf{rf}} r$), which is later overwritten by w in t ($w_r \xrightarrow{\mathsf{mo}} w$ and $(w, r) \in \mathsf{sthd}$), i.e. $r \xrightarrow{\mathsf{rb}_{\mathsf{b}}} w$, then w must have been issued after r, as otherwise r should have read from the later w and not w_r. Note that this is not the case for $\mathsf{rb} \setminus \mathsf{rb}_{\mathsf{b}}$: r can be issued *after* a later write w and still not observe it because the effect of w has not yet reached the memory (w is in $\mathbf{wb_L}/\mathbf{wb_R}$ or the store buffer of another thread).

The $\mathsf{rf}_{\overline{\mathsf{b}}}$ component in ob states that if a read r reads from a write w that passed through a different buffer (i.e. either w is part of an RDMA operation and went through $\mathbf{wb_L}/\mathbf{wb_R}$, or w is a CPU write by another thread and thus went through a different store buffer), then r can only read from w once it has reached the memory, i.e. only once w is observable, and thus w must have become observable before r. The $[\mathtt{nlW}]; \mathsf{pf}$ component states that if p polls from a NIC local write w, then w must have left the $\mathbf{wb_L}$ buffer and reached the memory. Note that this is not the case for `nrW` events: polling an `nrW` event w succeeds when w is in $\mathbf{wb_R}$ and w may not have yet reached the memory. The nfo in ob can be justified as in the case of ib.

The rb component ensures that a read r on location x observes the latest write on x that has reached the memory: if $w_r \xrightarrow{\mathsf{rf}} r$ and $w_r \xrightarrow{\mathsf{mo}} w$, i.e. $r \xrightarrow{\mathsf{rb}} w$, then w must have reached the memory after r was issued/observed, as otherwise r should have read from this later w and not w_r. As mo describes the order in which the writes on each location reach the memory, it is included in ob.

The third condition Def. 4.4 asks that $([\mathtt{Inst}]; \mathsf{ib}; \mathsf{ob})^+$ be irreflexive. Intuitively, if $\mathsf{e}_1 \xrightarrow{\mathsf{ib}} \mathsf{e}_2 \xrightarrow{\mathsf{ob}} \mathsf{e}_3$, then (1) e_1 is issued before e_2; (2) e_2 is issued before it is observed; and (3) e_2 is observed before e_3. If e_1 and e_3 are instantaneous events (i.e. their issue and observation points coincide) then e_1 necessarily precedes e_3, and thus the semantics prohibits creating a cycle from these dependencies.

Finally, we show that our RDMA$^{\text{TSO}}$ declarative semantics is equivalent to the simplified operational semantics in §3, with the full proof given in the extended version. Note that as a corollary (of theorem 3.1), our declarative semantics is also equivalent to the concrete operational semantics in §3.

THEOREM 4.5. *The RDMA$^{\text{TSO}}$ declarative semantics is equivalent to the simplified operational one.*

Recall that RDMA$^{\text{TSO}}$ assumes x86-TSO CPUs (subject to TSO consistency). As such, RDMA$^{\text{TSO}}$ can be seen as an extension of TSO [Alglave et al. 2014]. That is, as we show in the following theorem, for programs without remote operations, RDMA$^{\text{TSO}}$ and TSO coincide (see the extended version for the full proof).

THEOREM 4.6. *For programs without remote operations, TSO- and RDMA$^{\text{TSO}}$-consistency coincide.*

Declarative Semantics without PCIe. To relax the PCIe-specific constraint (stipulating that NIC local/remote reads propagates all pending NIC writes in $\mathbf{wb}_{\mathbf{L}}/\mathbf{wb}_{\mathbf{R}}$ to memory), we adapt our declarative model as follows: (1) we no longer need the nfo relation in Def. 4.2 and Def. 4.3 (as nfo is used to capture the 'NIC flush order' under PCIe); (2) we replace sqp in cells G8 and G9 of the oppo table (Fig. 9) with ✗ (as a later nrR/nlW no longer flushes $\mathbf{wb}_{\mathbf{R}}$ and thus may not observe an earlier nrW); and (3) we redefine rf_b and rb_b as follows to include events on the same queue pair:

$$\mathsf{rf}_\mathsf{b} \triangleq ([\mathtt{lW}];(\mathsf{rf}\cap\mathsf{sthd});[\mathtt{lR}]) \cup (\mathsf{rf}\cap\mathsf{sqp}) \qquad \mathsf{rb}_\mathsf{b} \triangleq ([\mathtt{lR}];(\mathsf{rb}\cap\mathsf{sthd});[\mathtt{lW}]) \cup (\mathsf{rb}\cap\mathsf{sqp})$$

Recall that $w \xrightarrow{\mathsf{rf}_\mathsf{b}} r$ (resp. $r \xrightarrow{\mathsf{rb}_\mathsf{b}} w$) denotes r reads from (resp. before) w before w is observable. In the strong model with PCIe guarantee, NIC local/remote reads flush the $\mathbf{wb}_{\mathbf{L}}/\mathbf{wb}_{\mathbf{R}}$, and thus r reads from (resp. before) w only once w is observable. By contrast, in the relaxed model without PCIe guarantee, NIC local/remote reads no longer flush the $\mathbf{wb}_{\mathbf{L}}/\mathbf{wb}_{\mathbf{R}}$, and thus r can read from (resp. before) w while it is still in $\mathbf{wb}_{\mathbf{L}}/\mathbf{wb}_{\mathbf{R}}$ (i.e. not yet observable). We thus expand the rf_b (resp. rb_b) definition to account for the possibility of reading from (resp. before) a write before it is observable.

5 Validating the RDMA$^{\text{TSO}}$ Model

To complement our formal semantics, we conducted an extensive validation of litmus tests on two distinct setups: InfiniBand (IB) and RDMA over Converged Ethernet (RoCE). In the IB setup, we used Dell PowerEdge R740 x86-64 machines, Intel(R) Xeon(R) Gold 6132 CPUs (2.60GHz, 14 cores), Ubuntu 22.04.2 LTS with Mellanox Technologies MT28908 Family [ConnectX-6] controller. In the RoCE setup, we used machines with Intel(R) Xeon(R) E-2286G CPUs (4.00GHz, 14 cores), ran Linux kernel version 4.18.0, Red Hat Enterprise Linux version 8 (477.27.1), with a Mellanox Technologies MT27800 Family [ConnectX-5] controller. Each of our tests is written in C – see the extended version and the code in the supplementary material [Ambal et al. 2024b].

Litmus Tests and Outcomes. We focused our validation efforts on a set of 37 litmus tests representative of a wide range of allowed and disallowed weak behaviours, including several variants of well-known concurrent tests in the literature such as 'store buffering' (SB), 'message passing' (MP), 'load buffering' (LB), 'independent reads of independent writes' (IRIW), parallel writes (2+2W) and so forth. Our results corroborate our RDMA$^{\text{TSO}}$ model and confirm that (1) RDMA$^{\text{TSO}}$ is *not too strong* (i.e. it does not prohibit behaviours exhibited by the hardware); and (2) RDMA$^{\text{TSO}}$ is *not too weak* (i.e. it does not admit too many weak behaviours not exhibited by hardware). Indeed, despite extensive testing we detected *no behaviours prohibited by RDMA$^{\text{TSO}}$*; and we observed *almost all behaviours allowed* by RDMA$^{\text{TSO}}$, with a few exceptions detailed below.

Bringing about Weak Behaviours. As with litmus testing for local (CPU-only) concurrency for established models such as TSO, observing a weak behaviour relies on several factors, including the order in which threads are scheduled and interleaved, as well as the timing of how writes are propagated and made visible to different threads (e.g. when writes are removed from the store buffer and propagated through the cache hierarchy and the memory). In the context of (RDMA) programs with remote operations, there are additional factors at play such as the NIC workload. As such, inducing weak behaviours necessitates creating suitable conditions, which may involve stressing the NIC and/or the CPU in the background. Indeed, as shown by the work of Dan et al.

[2016], this is far from straightforward as they failed to observe *any* weak behaviours at all. To remedy this, we used several techniques to elicit the weak behaviours permitted by the specification. Specifically, since no testing tool currently exists for automatically overwhelming or decelerating an RDMA system, in close consultation with NVIDIA experts we devised several techniques for creating bottlenecks and fostering the emergence of weak behaviours, as outlined below:

- **NIC delays**: We introduced delays on the Network Interface Card (NIC) by initiating numerous flood RDMA read or write operations between the client and the server on the designated queue pair (QP) at a specific juncture within the litmus test. As these operations still obey the same ordering rules, they cause some of the other operations in the litmus test to be delayed until they are successfully completed.
- **CPU delays**: We strategically injected CPU delays of random length into the litmus tests at certain points. We did this by selecting a random duration from three possibilities: 0 nanoseconds, half of the *round-trip time* (RTT) and the full RTT, delaying the CPU execution by the chosen duration. RTT represents the round-trip time between the client and the server, capturing the duration it takes for a packet to travel from the sender to the receiver and back. We implemented the delay functionality using a busy-wait loop, where the CPU remains occupied, repeatedly polling the clock without executing additional tasks. Once the delay duration has elapsed, the loop concludes, allowing the CPU to resume normal execution. While such delays can bring about weak behaviours in certain tests, such as that in Fig. 2d, it may impede observing others such as that in Fig. 2b, which requires no delay between $z^2 := x$ and $x := 1$. Therefore, by incorporating random delays, we aimed to encompass the full spectrum of potential scenarios, ensuring comprehensive coverage of behavioural variations across different tests. Interestingly, we observed that even minimal delays introduced by printing variables for debugging purposes can disrupt the manifestation of behaviours in certain tests (e.g. that in Fig. 2b), hence we limited our use of print statements.
- **High RDMA traffic loads**: While introducing NIC delays as described above can sometimes help induce certain interleavings, e.g. by delaying one node while letting another to continue, in some cases delays do not help as both relevant operations are delayed by the same NIC delay operation. In such cases, using *background traffic* on unrelated queue pairs allows additional interleavings. Specifically, we generated concurrent high RDMA traffic loads on the background using numerous queue pairs and utilising the zero-copy mode (transferring data directly between the memory of the sender and receiver without intermediate copying). This increased RDMA activity introduces competition for shared system resources such as CPU cycles, memory bandwidth, network capacity and NIC processing capacity. This led to contention amongst threads in a litmus test, potentially altering their scheduling behaviour. This strategic approach proved pivotal in uncovering weak behaviours in several tests such as **MP3bis** in the extended version, which might otherwise have gone unnoticed. We observed that the number of queue pairs utilised to generate the traffic load plays a crucial role in certain tests. For instance, in the case of **GFP2** (see the extended version), we used 128 queue pairs to induce the weak behaviour.
- **Synchronisation**: As is common practice in validation via litmus testing, we employed loops to bring about the desired weak behaviours. For instance, in order to ensure that a local (resp. remote) load operation reads the desired value required by a given weak behaviour, we place the operation within a loop (in effect replacing the single operation with multiple such operations), iterating until the desired value is read. In other cases, we placed the entire litmus test within a loop. Using loops this way can sometimes replace the CPU delay mechanism discussed above, allowing it to pinpoint the delay needed to reproduce the behaviour.

Validation Results. Using the techniques above, we successfully confirmed almost all weak behaviours allowed by RDMA$^{\text{TSO}}$, including all ✓ examples in Figs. 1 to 3, as well as several variants of MP, SB, LB, IRIW, 2+2W, and others, observed at varying frequencies (see the extended version for more details). The frequency of observing these behaviours is influenced not only by the nuances of the litmus test itself, but also by the infrastructure settings such as the network setup, switch configuration and NIC capabilities. Moreover, dynamic and unpredictable factors such as network congestion, packet drops, latency fluctuations, bandwidth utilisation and routing changes also significantly impact the observations. An interesting phenomenon we encountered was the variability in behaviour manifestation even within a specific test scenario. For example, in the case of **GFP2**, the weak behaviour was observed at one of our test premises but not the other (we had two test premises at two distinct geographical locations). This observation highlights the intricacy and subtlety involved in inducing weak behaviours.

In 4 of the litmus tests, we did not observe the allowed weak behaviour because the hardware implementation did not utilise the weakness permitted by the specification. As discussed in §2, manifesting the weak behaviour in such cases relies on polling a put operation before the remote NIC write completes, e.g. as in Fig. 3d. This weak behaviour is allowed by RDMA$^{\text{TSO}}$ because the specification allows the remote NIC to return an acknowledgement *before* performing the associated write. Nevertheless, according to NVIDIA experts, this weakness is not utilised by the hardware implementations. To the best of our knowledge, these weak behaviours cannot be observed on current hardware implementations, but they might emerge in future ones.

Limitations. The main limitation of our validation is that weak behaviours are only exposed by hand-crafted techniques (discussed above) that stress the system in certain ways. This is currently a difficult trial-and-error process that requires a high degree of knowledge of current hardware implementations (which we acquired through close consultation with NVIDIA experts). As such, executing RDMA tests is currently not amenable to *mass automation* as in the frameworks of [Alglave et al. 2021, 2014; Raad et al. 2022]. To adapt these frameworks for RDMA, one would need to develop systematic heuristics for automatically applying the techniques discussed above. We leave this challenge to future work.

6 Related and Future Work

RDMA Semantics. To our knowledge, the only existing work on the formal semantics of RDMA programs is that of coreRMA [Dan et al. 2016], which has several key limitations, as follows. (1) Although Dan et al. [2016] attempted to validate coreRMA, they observed *none* of the weak behaviours allowed by coreRMA in existing hardware implementations. This is in contrast to our work, where we have observed almost all weak behaviours allowed by RDMA$^{\text{TSO}}$ in existing implementations, and in the rare cases where we could not observe a behaviour, we have confirmed that this is because existing implementations explicitly did not utilise the weakness allowed by the specification. (2) coreRMA assumes that CPU concurrency is governed by the strong and unrealistic SC model [Lamport 1979]. (3) coreRMA only presents a declarative (and not operational) characterisation. (4) Most importantly, coreRMA departs from the RDMA specification [IBTA 2022] in three important ways, as follows.

First, they do not model the poll operation, `poll`(n), described in the specification. Instead, they model a flush operation, `flush`(n), whose behaviour does not match any operation defined in the specification. Specifically, `flush`(n) waits for all previous RDMA operations on the n-queue pair to complete and further blocks later CPU operations and RDMA operations on the *same* queue pair. In other words, `flush`(n) is tantamount to `rfence`(n) followed by `mfence`. However, `flush`(n) does *not* block later RDMA operations on *different* queue pairs, and hence under coreRMA there is no clean way to enforce an order on two RDMA operations on different queue pairs.

For instance, to ensure that $x := y^2; z^3 := x$ is executed in order (so that z gets the updated value of x), it is not enough to add a flush and write $x := y^2; \underline{\texttt{flush(2)}}; z^3 := x$. Instead, one must also add a superfluous *CPU operation* and write e.g. $x := y^2; \underline{\texttt{flush(2)}; a := w}; z^3 := x$. This is because $\texttt{flush(2)}$ blocks the CPU operation $a := w$, which in turn blocks $z^3 := x$.

Second, RDMA$^{\text{TSO}}$ preserves the order between two NIC local reads and two NIC local writes (cells F6 and I9 in Fig. 9), while coreRMA does not, violating the RDMA specification [IBTA 2022]. For instance, given $y^2 := x; z^2 := w$, RDMA$^{\text{TSO}}$ guarantees x is read before w, while there is no such guarantee under coreRMA; i.e. in such scenarios coreRMA is *weaker* than the specification.

Finally, as per the RDMA specification, under RDMA$^{\text{TSO}}$ remote reads can be reordered with respect to other remote operations on the same queue pair (cells H7 and H8), while under coreRMA they cannot; i.e. in such scenarios coreRMA is *stronger* than the specification. Consequently, as coreRMA is weaker than the specification in some scenarios and stronger in others, it is *neither a strict abstraction, nor a strict refinement* of the specification.

Weak Memory Models. Existing literature includes several examples of weak consistency models, both at hardware and software levels. On the hardware side, several works have formalised the semantics of the x86 architecture [Abdulla et al. 2015; Alglave et al. 2014; Raad et al. 2022; Sewell et al. 2010]. However, none of these works covered the consistency semantics of RDMA programs in the context of x86 machines. Similarly, several works have formalised the semantics of the ARMv8 and POWER architectures, both operationally and declaratively [Alglave et al. 2021, 2014; Chakraborty and Vafeiadis 2019; Flur et al. 2016; Mador-Haim et al. 2012; Pulte et al. 2018; Sarkar et al. 2011]. On the software side, there has been a number of formal models for C11 consistency [Batty et al. 2011; Kang et al. 2017; Lahav et al. 2016, 2017; Lee et al. 2020; Nienhuis et al. 2016; Pichon-Pharabod and Sewell 2016] with verified compilation schemes [Moiseenko et al. 2020; Podkopaev et al. 2017, 2019], Java [Bender and Palsberg 2019; Manson et al. 2005], transactional memory [Raad et al. 2018, 2019a; Xiong et al. 2020], the Linux kernel [Alglave et al. 2018] and the ext4 filesystem [Kokologiannakis et al. 2021]. 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 [Cho et al. 2021; Khyzha and Lahav 2021; Raad and Vafeiadis 2018; Raad et al. 2020b, 2019b], as well as program logics for verifying such programs [Bila et al. 2022; Raad et al. 2020a].

Future Work. We plan to build over our work presented here in several ways. First, we aim to adapt existing methods for automatically generating litmus tests (e.g. [Alglave et al. 2021, 2014]) to RDMA programs by developing heuristics for automatically applying the inducement/stressing techniques discussed in §5 for bringing about weak behaviours. Second, we plan to formalise the semantics of RDMA programs in the context of the ARMv8 hardware architecture (i.e. when CPU concurrency is governed by ARMv8 rather than TSO). Third, we plan to verify existing RDMA implementations such as those of the Verbs [linux-rdma 2018] and libfabric [OpenFabrics 2016] APIs. Lastly, using our formal RDMA$^{\text{TSO}}$ semantics, we plan to develop *manual* reasoning techniques such as program logics (underpinned by our operational semantics), as well as *automated* verification techniques such as model checking (based on our declarative semantics) for RDMA.

Acknowledgments

We would like to thank Alexey Gotsman, Adam Morrison, Noam Rinetzky, and especially Yuri Meshman, for starting this research and whose preliminary memory model underpins the results presented here. Additional thanks to Yamin Friedman, Daniel Marcovitch and Liran Liss for sharing their insights into the IBTA specifications and NVIDIA's RDMA implementations. We also thank the anonymous reviewers for their valuable feedback and Viktor Vafeiadis for many fruitful discussions. Guillaume Ambal is supported by the EPSRC grant EP/X037029/1. Brijesh Dongol is supported by VeTSS and EPSRC projects EP/Y036425/1, EP/X037142/1, EP/X015149/1, EP/V038915/1, and EP/R025134/2. Ori 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 number 814/22). Azalea Raad is supported by a UKRI fellowship MR/V024299/1, by the EPSRC grant EP/X037029/1, and by VeTSS.

References

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. 2021. Deciding Reachability under Persistent X86-TSO. *Proc. ACM Program. Lang.* 5, POPL, Article 56 (Jan. 2021), 32 pages. https://doi.org/10.1145/3434337

Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan-Phong Ngo. 2015. 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*. Springer-Verlag New York, Inc., New York, NY, USA, 308–332. https://doi.org/10.1007/978-3-662-46669-8_13

Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, and Igor Zablotchi. 2019. The Impact of RDMA on Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, Peter Robinson and Faith Ellen (Eds.). ACM, 409–418. https://doi.org/10.1145/3293611.3331601

Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2 (2021), 8:1–8:54. https://doi.org/10.1145/3458926

Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2018. Frightening Small Children and Disconcerting Grown-Ups: Concurrency in the Linux Kernel. *SIGPLAN Not.* 53, 2 (March 2018), 405–418. https://doi.org/10.1145/3296957.3177156

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

Guillaume Ambal, Brijesh Dongol, Haggai Eran, Vasileios Klimis, Ori Lahav, and Azalea Raad. 2024a. Extended Version. https://www.soundandcomplete.org/papers/OOPSLA2024/RDMA/rdma-extended.pdf

Guillaume Ambal, Brijesh Dongol, Haggai Eran, Vasileios Klimis, Ori Lahav, and Azalea Raad. 2024b. Project page for Semantics of Remote Direct Memory Access. https://www.soundandcomplete.org/papers/OOPSLA2024/RDMA

Don Anderson. 1999. *FireWire system architecture (2nd ed.): IEEE 1394a.* Addison-Wesley Longman Publishing Co., Inc., USA.

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

John Bender and Jens Palsberg. 2019. A Formalization of Java's Concurrent Access Modes. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 142 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360568

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

M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. 2015. Intel Omni-Path Architecture: Enabling scalable, high performance fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI) (HOTI 2015)*. 1–9. https://doi.org/10.1109/HOTI.2015.22

Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and Enforcing Robustness against TSO. In *ESOP 2013 (LNCS, Vol. 7792)*. Springer, 533–553. https://doi.org/10.1007/978-3-642-37036-6_29

Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding Thin-Air Reads with Event Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 70 (Jan. 2019), 28 pages. https://doi.org/10.1145/3290383

Kyeongmin Cho, Sung-Hwan Lee, Azalea Raad, and Jeehoon Kang. 2021. 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* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 16–31. https://doi.org/10.1145/3453483.3454027

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

D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A.M. Merritt, E. Gronke, and C. Dodd. 1998. The Virtual Interface Architecture. *IEEE Micro* 18, 2 (1998), 66–76. https://doi.org/10.1109/40.671404

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 608–621. https://doi.org/10.1145/2837614.2837615

Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. 2018. Enabling Highly Scalable Remote Memory Access Programming with MPI-3 One Sided. *Commun. ACM* 61, 10 (sep 2018), 106–113. https://doi.org/10.1145/3264413

IBTA. 2022. InfiniBand Architecture Specification Volume 1 Release 1.6. https://www.infinibandta.org/ibta-specification/.

InfiniBand Trade Association (IBTA). 2018. The RoCE Initiative. https://www.infinibandta.org/roce-initiative/ (Accessed: July 2023).

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. *SIGPLAN Not.* 52, 1 (Jan. 2017), 175–189. https://doi.org/10.1145/3093333.3009850

Artem Khyzha and Ori Lahav. 2021. Taming X86-TSO Persistency. *Proc. ACM Program. Lang.* 5, POPL, Article 47 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434328

Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, and Viktor Vafeiadis. 2021. PerSeVerE: Persistency Semantics for Verification under Ext4. *Proc. ACM Program. Lang.* 5, POPL, Article 43 (jan 2021), 29 pages. https://doi.org/10.1145/3434324

Ori Lahav and Udi Boker. 2020. Decidable verification under a causally consistent shared memory. In *PLDI 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 211–226. https://doi.org/10.1145/3385412.3385966

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming Release-Acquire Consistency. *SIGPLAN Not.* 51, 1 (Jan. 2016), 649–662. https://doi.org/10.1145/2914770.2837643

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: Global Optimizations in Relaxed Memory Concurrency. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 362–376. https://doi.org/10.1145/3385412.3386010

linux-rdma. 2018. RDMA core. https://github.com/linux-rdma/rdma-core/ (Accessed: Jul. 2023).

Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. 2012. An Axiomatic Memory Model for POWER Multiprocessors. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 495–512. https://doi.org/10.1007/978-3-642-31424-7_36

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 378–391. https://doi.org/10.1145/1040305.1040336

Evgenii Moiseenko, Anton Podkopaev, Ori Lahav, Orestis Melkonian, and Viktor Vafeiadis. 2020. Reconciling Event Structures with Modern Multiprocessors (Artifact). *Dagstuhl Artifacts Series* 6, 2 (2020), 4:1–4:3. https://doi.org/10.4230/DARTS.6.2.4

Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. 2016. 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* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 111–128. https://doi.org/10.1145/2983990.2983997

Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/2541940.2541965

NVIDIA Corporation. 2021. NVIDIA BlueField-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf (Accessed: Jul. 2023).

OpenFabrics. 2016. RDMA core. https://ofiwg.github.io/libfabric/ (Accessed: Jul. 2023).

PCI-SIG. 2022. PCI Express Base Specification Revision 6.0 Version 1.0. https://pcisig.com/pci-express-6.0-specification.

Jean Pichon-Pharabod and Peter Sewell. 2016. 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* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 622–633. https://doi.org/10.1145/2837614.2837616

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2017. Promising Compilation to ARMv8 POP. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22:1–22:28. https://doi.org/10.4230/LIPIcs.ECOOP.2017.22

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the Gap Between Programming Languages and Hardware Weak Memory Models. *Proc. ACM Program. Lang.* 3, POPL, Article 69 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290382

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article 19 (Dec. 2018), 29 pages. https://doi.org/10.1145/3158107

Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: A Simpler and Faster Operational Concurrency Model. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3314221.3314624

Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2018. On Parallel Snapshot Isolation and Release/Acquire Consistency. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 940–967.

Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2019a. On the Semantics of Snapshot Isolation. In *Verification, Model Checking, and Abstract Interpretation*, Constantin Enea and Ruzica Piskac (Eds.). Springer International Publishing, Cham, 1–23.

Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2020a. Persistent Owicki-Gries Reasoning: A Program Logic for Reasoning about Persistent Programs on Intel-X86. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 151 (nov 2020), 28 pages. https://doi.org/10.1145/3428219

Azalea Raad, Luc Maranget, and Viktor Vafeiadis. 2022. Extending Intel-X86 Consistency and Persistency: Formalising the Semantics of Intel-X86 Memory Types and Non-Temporal Stores. *Proc. ACM Program. Lang.* 6, POPL, Article 22 (jan 2022), 31 pages. https://doi.org/10.1145/3498683

Azalea Raad and Viktor Vafeiadis. 2018. Persistence Semantics for Weak Memory: Integrating Epoch Persistency with the TSO Memory Model. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 137 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276507

Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020b. Persistency Semantics of the Intel-X86 Architecture. *Proc. ACM Program. Lang.* 4, POPL, Article 11 (Dec. 2020), 31 pages. https://doi.org/10.1145/3371079

Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019b. Weak Persistency Semantics from the Ground Up: Formalising the Persistency Semantics of ARMv8 and Transactional Models. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 135 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360561

Renato J. Recio, Paul R. Culley, Dave Garcia, Bernard Metzler, and Jeff Hilland. 2007. A Remote Direct Memory Access Protocol Specification. RFC 5040. https://doi.org/10.17487/RFC5040

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 175–186. https://doi.org/10.1145/1993498.1993520

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. https://doi.org/10.1145/1785414.1785443

Alexander Shpiner, Eitan Zahavi, Omar Dahley, Aviv Barnea, Rotem Damsker, Gennady Yekelis, Michael Zus, Eitan Kuta, and Dean Baram. 2017. RoCE Rocks without PFC: Detailed Evaluation. In *Proceedings of the Workshop on Kernel-Bypass Networks* (Los Angeles, CA, USA) *(KBNets '17)*. Association for Computing Machinery, New York, NY, USA, 25–30. https://doi.org/10.1145/3098583.3098588

SPARC. 1992. *The SPARC Architecture Manual: Version 8.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

S. Van Doren. 2019. Abstract - HOTI 2019: Compute Express Link. In *2019 IEEE Symposium on High-Performance Interconnects (HOTI) (HOTI 2019)*. 18–18. https://doi.org/10.1109/HOTI.2019.00017

Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 87–104. https://doi.org/10.1145/2815400.2815419

Shale Xiong, Andrea Cerone, Azalea Raad, and Philippa Gardner. 2020. Data Consistency in Transactional Storage Systems: A Centralised Semantics. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 21:1–21:31. https://doi.org/10.4230/LIPIcs.ECOOP.2020.21

Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 523–536. https://doi.org/10.1145/2785956.2787484

Received 2024-04-05; accepted 2024-08-18