Correctly Programming Remote Direct Memory Access (RDMA)

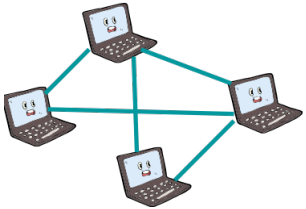
IFIP 2.3
Athens 2025

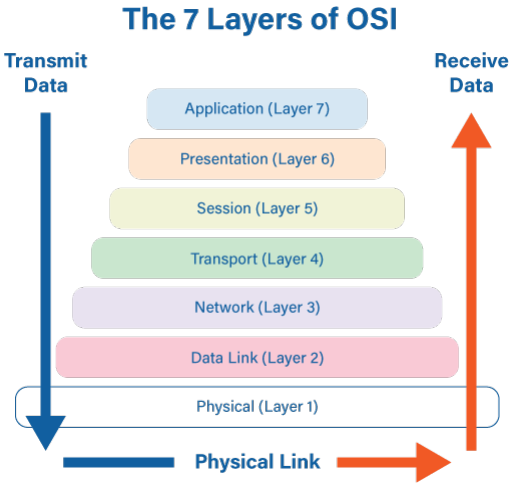
Brijesh Dongol[1]

with Guillaume Ambal[2], Gregory Chockler[1], Haggai Eran[3], Vasileios Klimis[4], Ori Lahav[5], Azalea Raad[2], Viktor Vafeiadis[6]

[1]University of Surrey, [2]Imperial College London, , [3]NVIDIA,
[4]Queen Mary University of London, [5]Tel Aviv University, [6]MPI-SWS

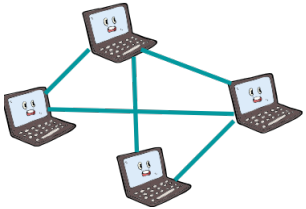
Problem: Slow communication (TCP/IP)

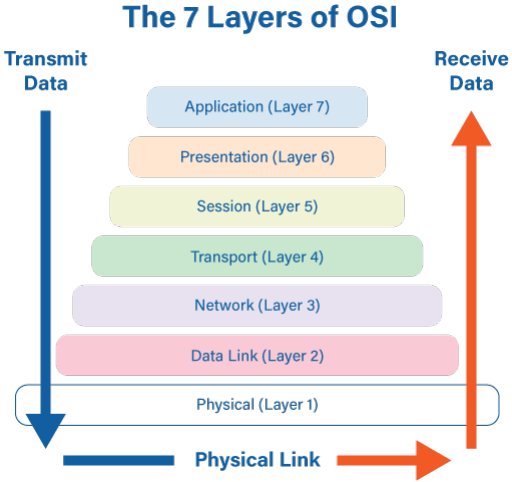




Latency : $\sim$ ms

Problem: Slow communication (TCP/IP)



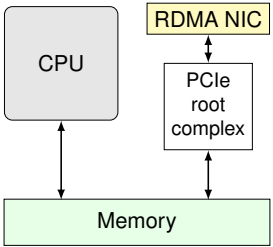


Too slow for modern distributed applications

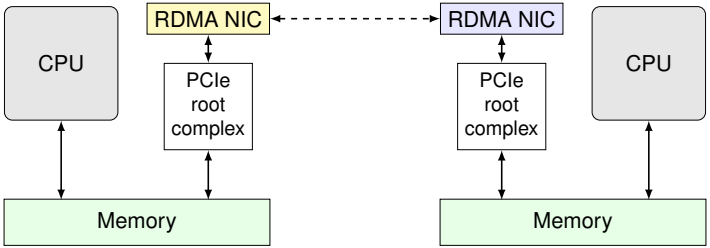
- datacenters
- cloud servers
- in-memory databases
- HPC systems
- distributed AI training / federated ML
- etc

Latency : $\sim$ ms

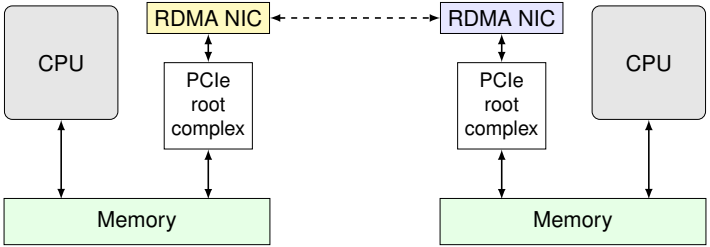
Remote Direct Memory Access (RDMA)



Remote Direct Memory Access (RDMA)

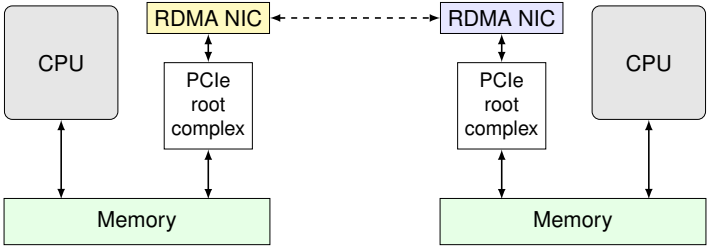


Remote Direct Memory Access (RDMA)



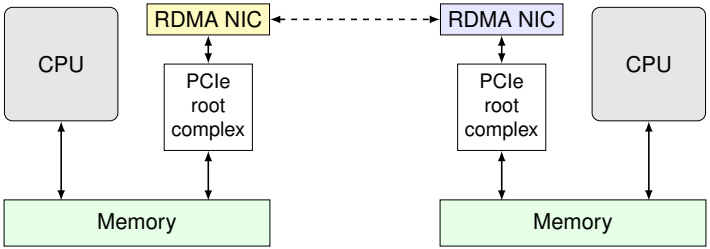
- Directly read from / write to remote memory
- Zero-copy kernel bypass
- Latency: $\sim \mu$s
- Recently becoming more widespread — Infiniband and RoCE

Remote Direct Memory Access (RDMA)



- Directly read from / write to remote memory
- Zero-copy kernel bypass
- Latency: $\sim \mu$s
- Recently becoming more widespread — Infiniband and RoCE
- But:
  - Complex semantics described via informal technical manuals
  - Buggy implementations

Remote Direct Memory Access (RDMA)



Our work: EPSRC project "*SACRED-MA: Safe And seCure REmote Direct Memory Access*"

- **Investigators**: Brijesh Dongol, Gregory Chockler (Surrey); Azalea Raad (Imperial);
- **Post Docs**: Guillaume Ambal (Imperial) and ??? (Surrey);
- **Partners**: NVIDIA (formerly Mellanox), Arm, Tel Aviv University, Cornell, Max Planck Institute, Uni. Colorado Boulder

Our work

Formalising RDMA semantics (assuming TSO for the CPU)

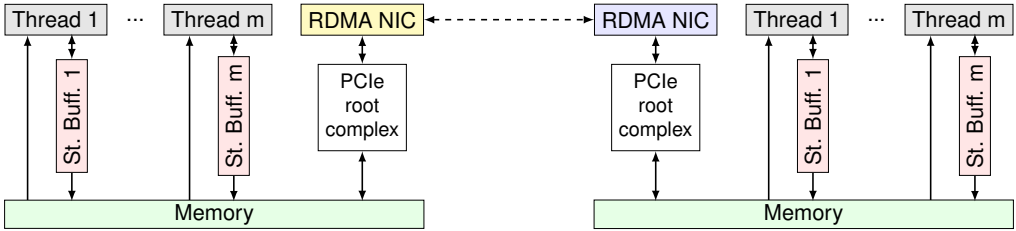
1. Operational semantics
2. Declarative semantics
3. Semantics proved equivalent
4. Empirical validation

} OOPSLA 2024: Ambal, Dongol, Eran, Klimis, Lahav, Raad

5. LOCO library verification

} Under submission: Ambal, Chockler, Dongol, Raad, Vafeiadis

RDMA^{TSO}: RDMA + Total Store Order (TSO)



Programming Model and Syntax

Model:

- network with several nodes
- each node can have several threads

Syntax

Mem Locs	$x, y, z, \ldots$	
Nodes	$n, n_1, n_2, n_3, \ldots$	
Commands	$c ::= \; x := e \mid r := x \mid \texttt{mfence} \mid \ldots$	$\leftarrow$ usual TSO
	$\mid x^n := y \mid x := y^n \mid \texttt{poll}(n) \mid \ldots$	$\leftarrow$ RDMA

Programming Model and Syntax

Model:

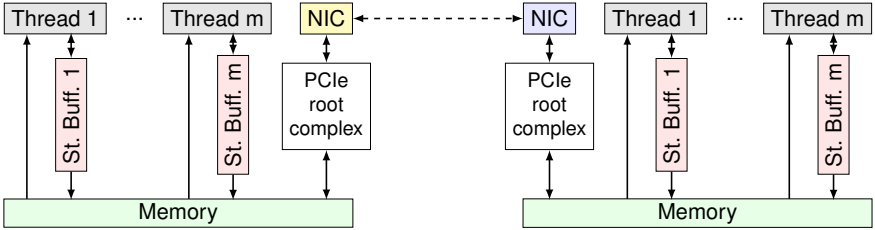
- network with several nodes
- each node can have several threads

Syntax

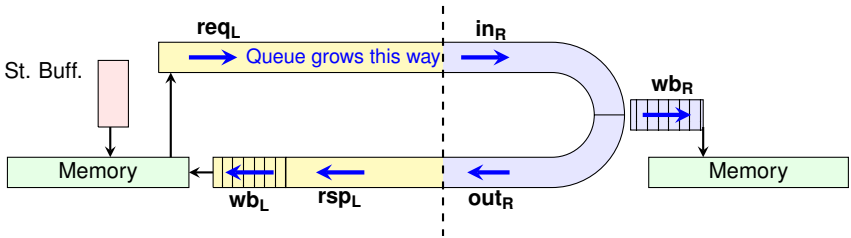
Mem Locs	$x, y, z, \ldots$	
Nodes	$n, n_1, n_2, n_3, \ldots$	
Commands	$c ::= \; x := e \mid r := x \mid \texttt{mfence} \mid \ldots$	$\leftarrow$ usual TSO
	$\mid x^n := y \mid x := y^n \mid \texttt{poll}(n) \mid \ldots$	$\leftarrow$ RDMA

Write $\widetilde{x} := y$ for $x^n := y$ when n is uniquely identifiable (similarly $x := \widetilde{y}$)

RDMA^{TSO} Architectures



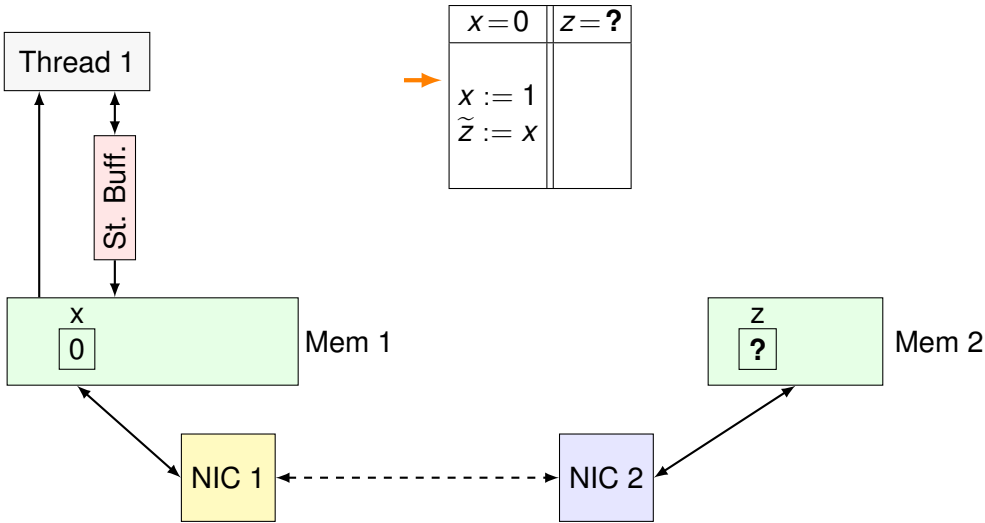
In detail, RDMA connection is through Queue Pairs from one thread to another

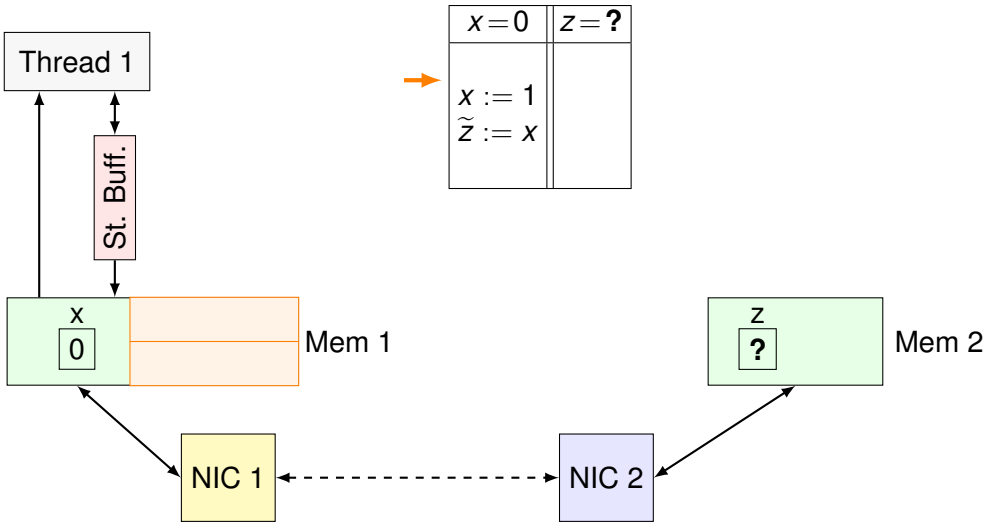


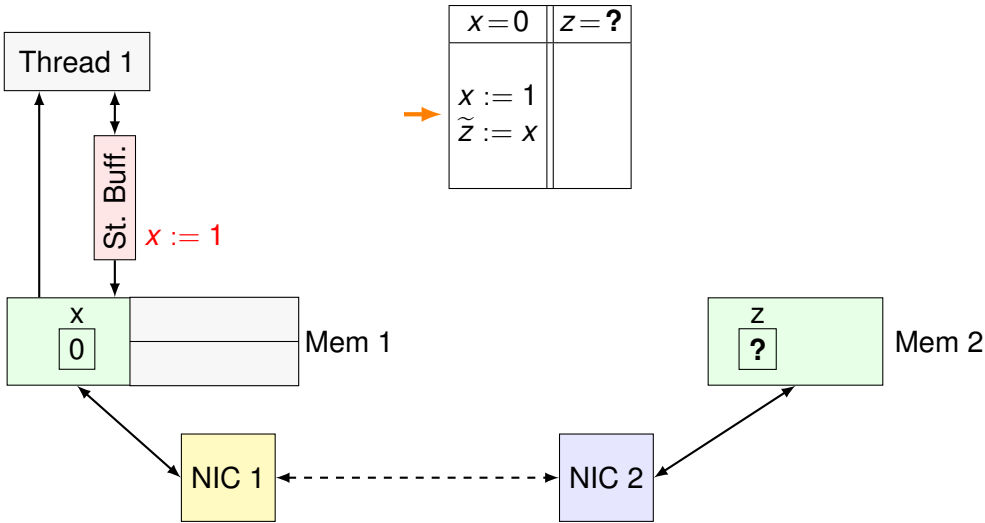
Example 1: Local write — Remote write

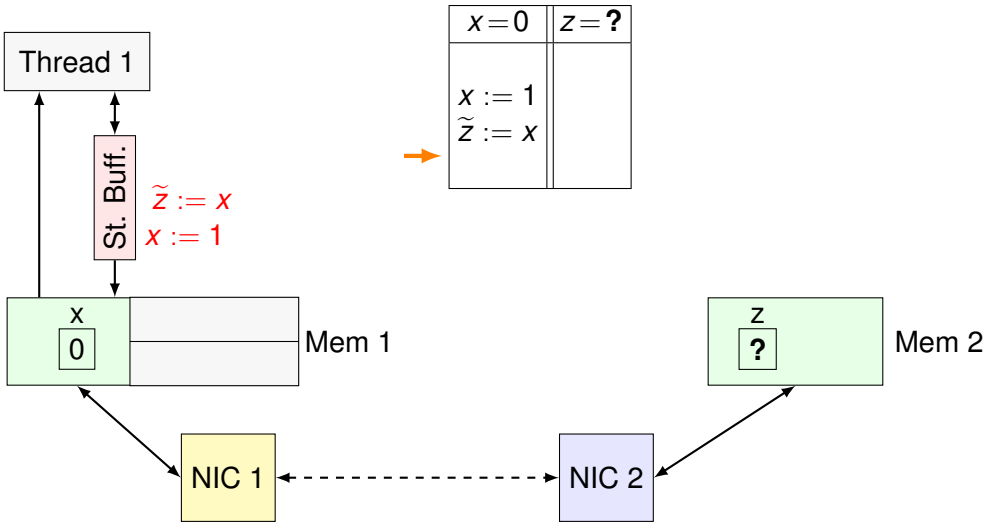
$x = 0$	$z = \mathbf{?}$
$x := 1$ $\widetilde{z} := x$	

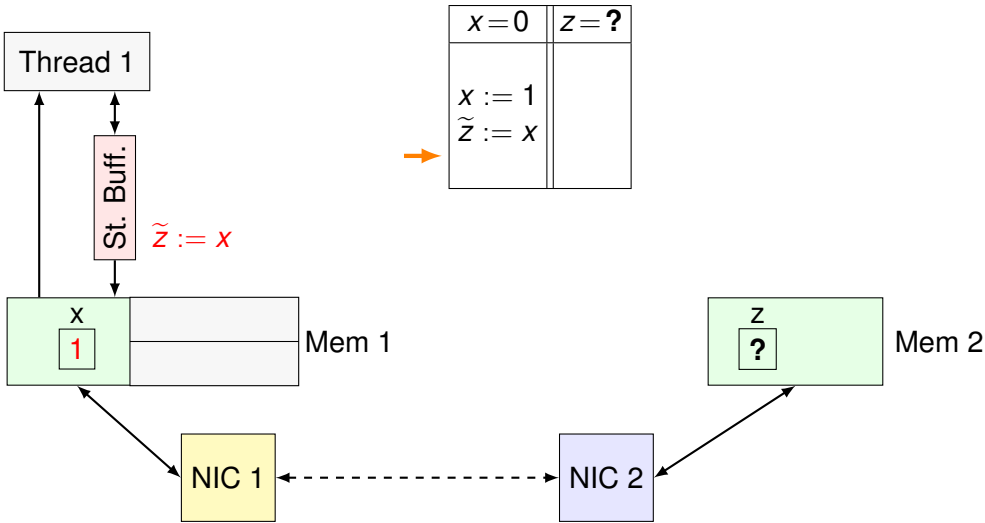
$z = \mathbf{?}$

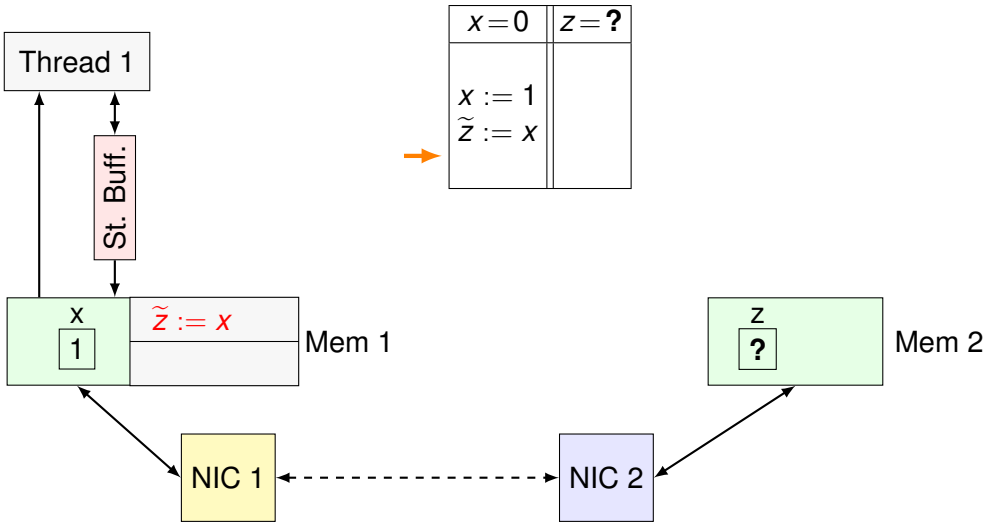

Thread 1
St. Buff.
x
0
Mem 1
z
?
Mem 2
NIC 1
NIC 2
x = 0
z = ?
x := 1
z̃ := x

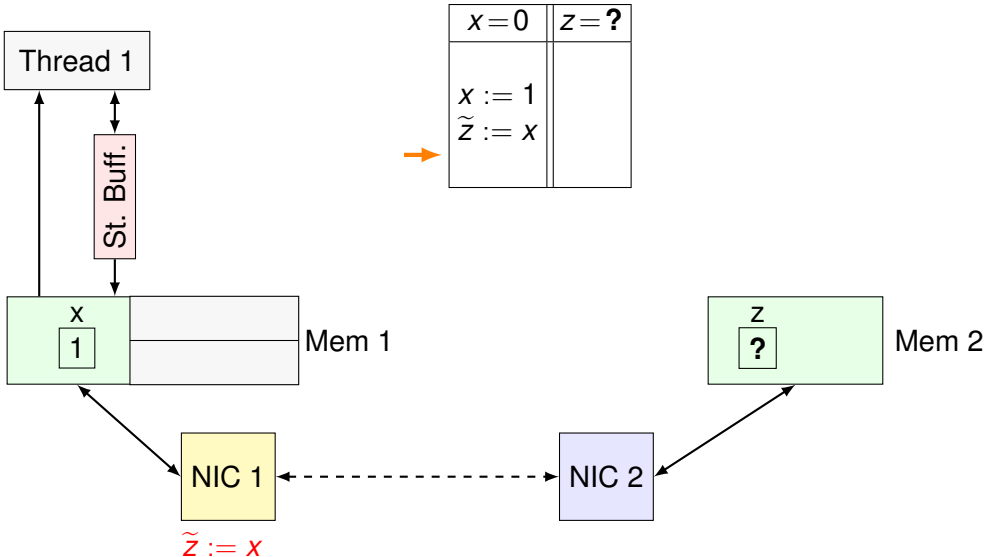

Thread 1
St. Buff.
x
0
Mem 1
z
?
Mem 2
NIC 1
NIC 2
x = 0
z = ?
x := 1
z̃ := x

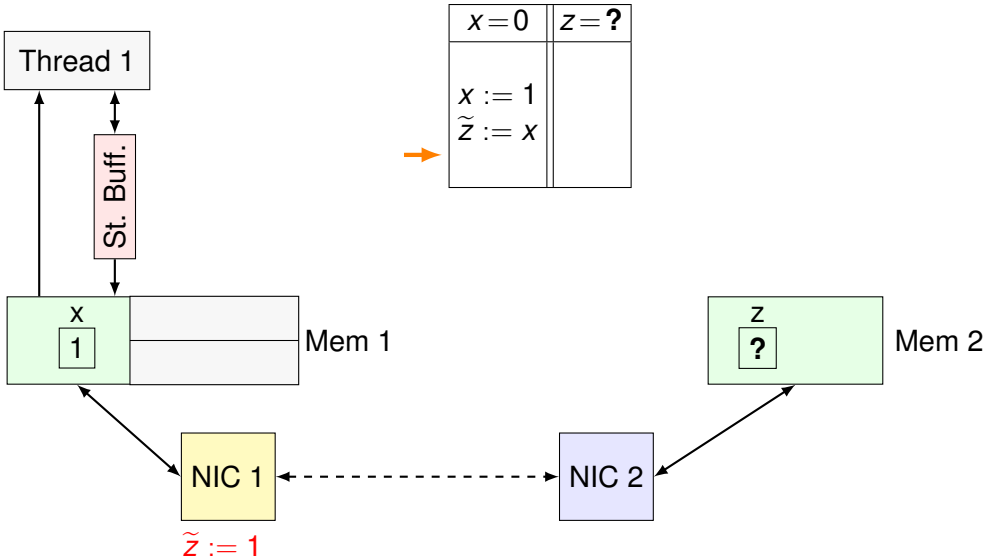

Thread 1
St. Buff.
$x := 1$
x
0
Mem 1
z
?
Mem 2
NIC 1
NIC 2
$x = 0$
$z = ?$
$x := 1$
$\widetilde{z} := x$

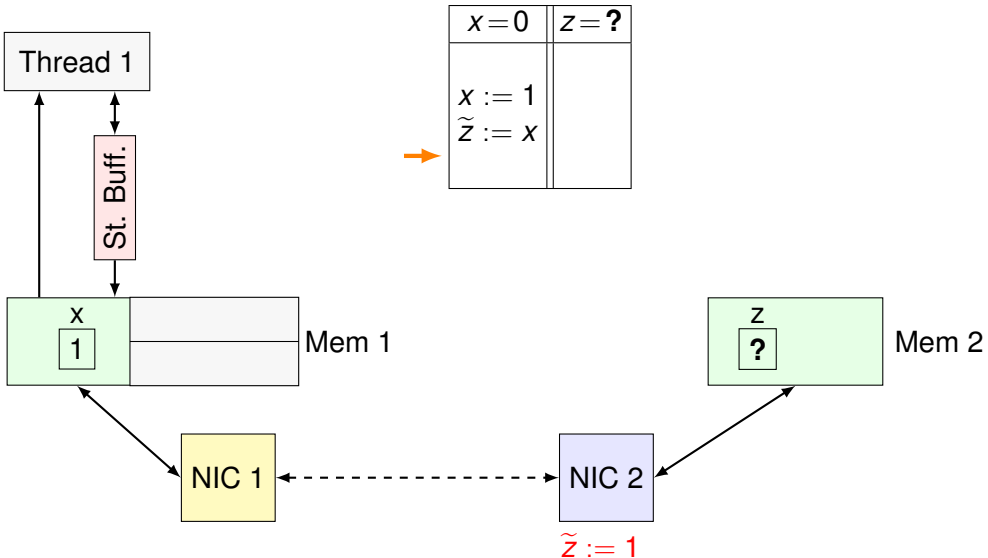

Thread 1
St. Buff.
$\widetilde{z} := x$
$x := 1$
$x = 0$
$z = ?$
$x := 1$
$\widetilde{z} := x$
x
0
Mem 1
z
?
Mem 2
NIC 1
NIC 2

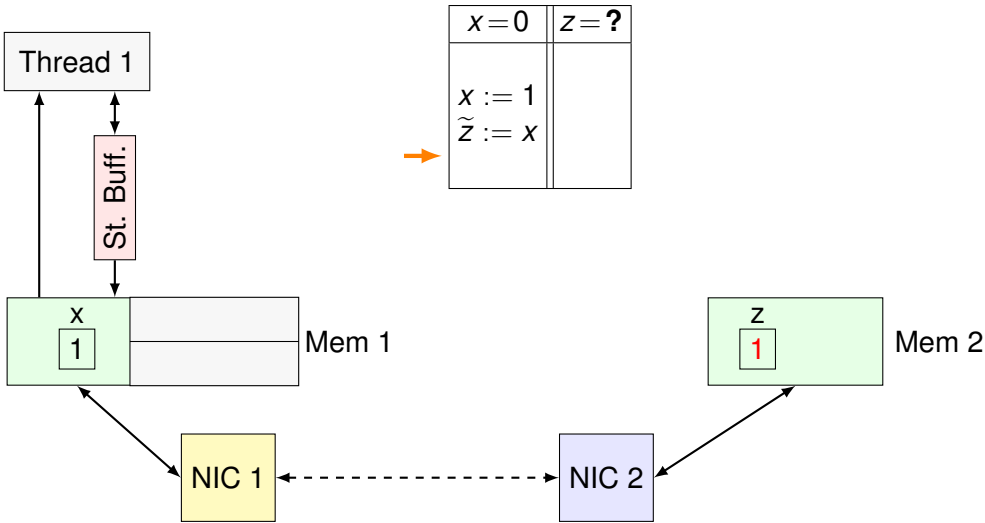

Thread 1
St. Buff.
$\widetilde{z} := x$
x
1
Mem 1
z
?
Mem 2
NIC 1
NIC 2
$x = 0$
$z = ?$
$x := 1$
$\widetilde{z} := x$

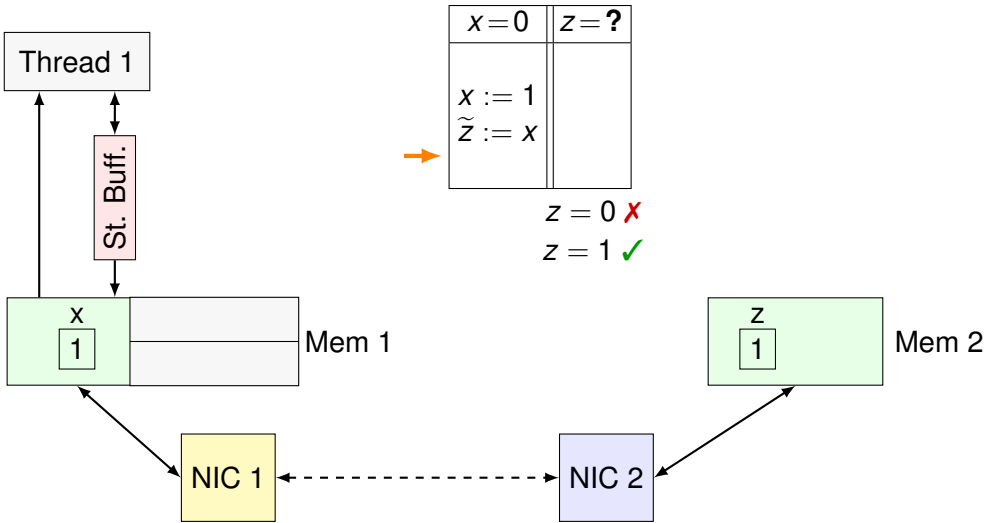

Thread 1
St. Buff.
x
1
$\widetilde{z} := x$
Mem 1
z
?
Mem 2
NIC 1
NIC 2
$x = 0$
$z = ?$
$x := 1$
$\widetilde{z} := x$


Thread 1
St. Buff.
x
1
Mem 1
z
?
Mem 2
NIC 1
NIC 2
$\widetilde{z} := x$
$x = 0$
$z = ?$
$x := 1$
$\widetilde{z} := x$


Thread 1
St. Buff.
x
1
Mem 1
z
?
Mem 2
NIC 1
NIC 2
$\widetilde{z} := 1$
$x = 0$
$z = ?$
$x := 1$
$\widetilde{z} := x$


Thread 1
St. Buff.
x
1
Mem 1
z
?
Mem 2
NIC 1
NIC 2
$x = 0$
$z = ?$
$x := 1$
$\widetilde{z} := x$
$\widetilde{z} := 1$

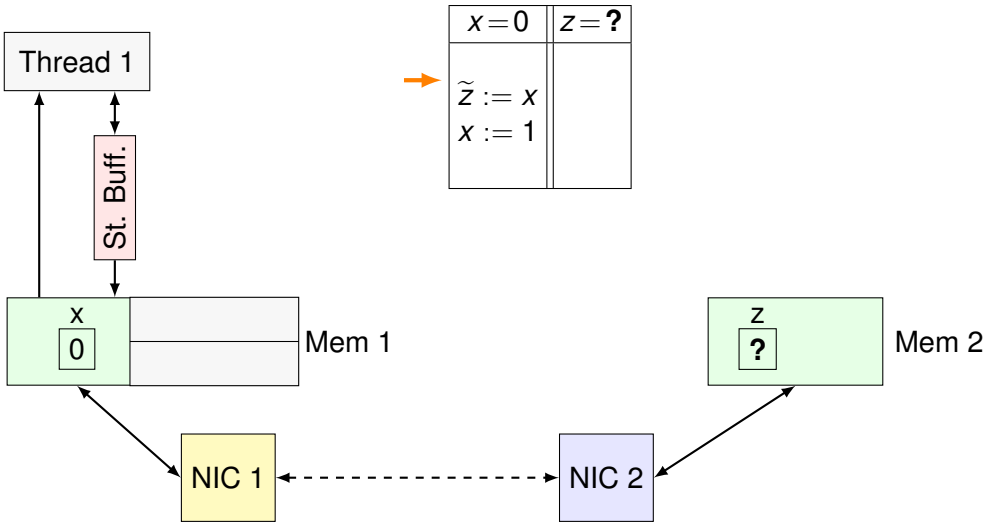

Thread 1
St. Buff.
x
1
Mem 1
z
1
Mem 2
NIC 1
NIC 2
x = 0
z = ?
x := 1
z̃ := x

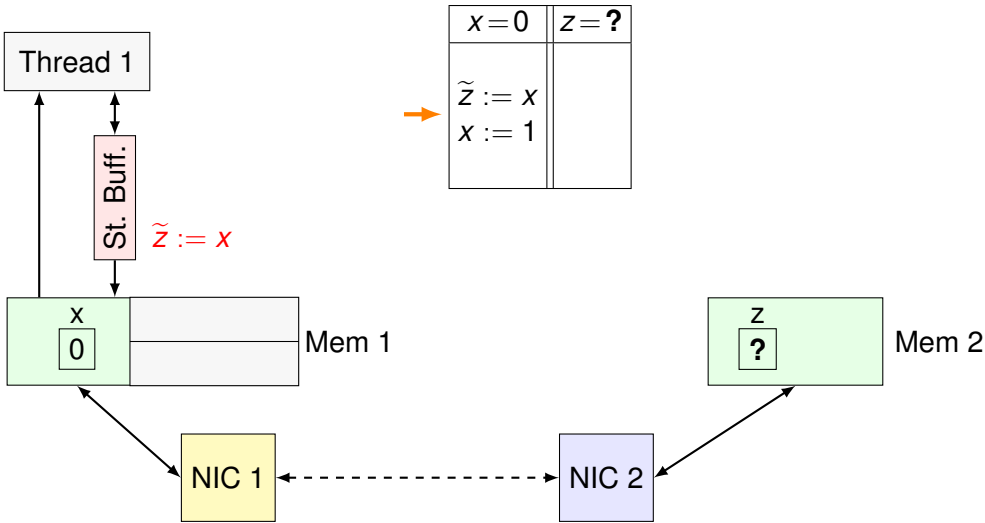

Thread 1
St. Buff.
x
1
Mem 1
z
1
Mem 2
NIC 1
NIC 2
x = 0
z = ?
x := 1
z̃ := x
z = 0 ✗
z = 1 ✓

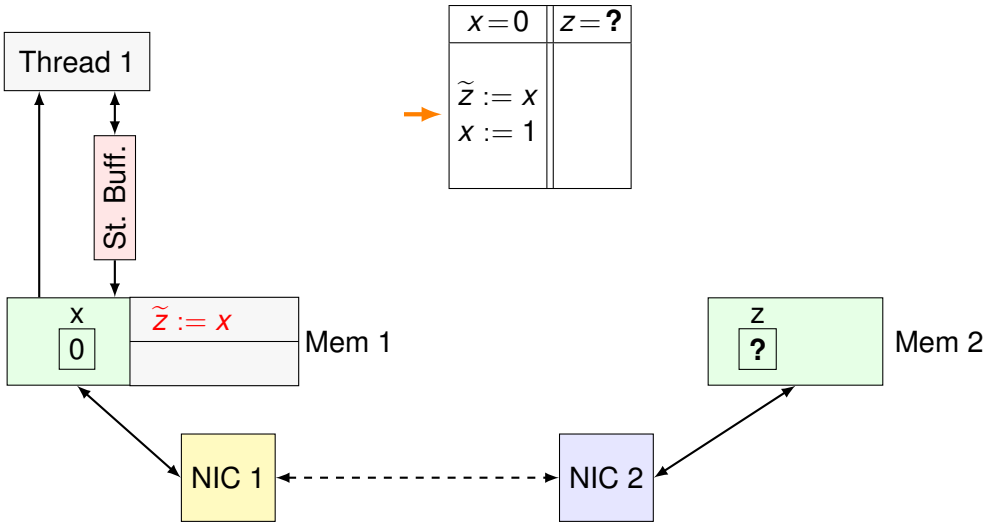
Example 2: Remote Write from Local – Local Write

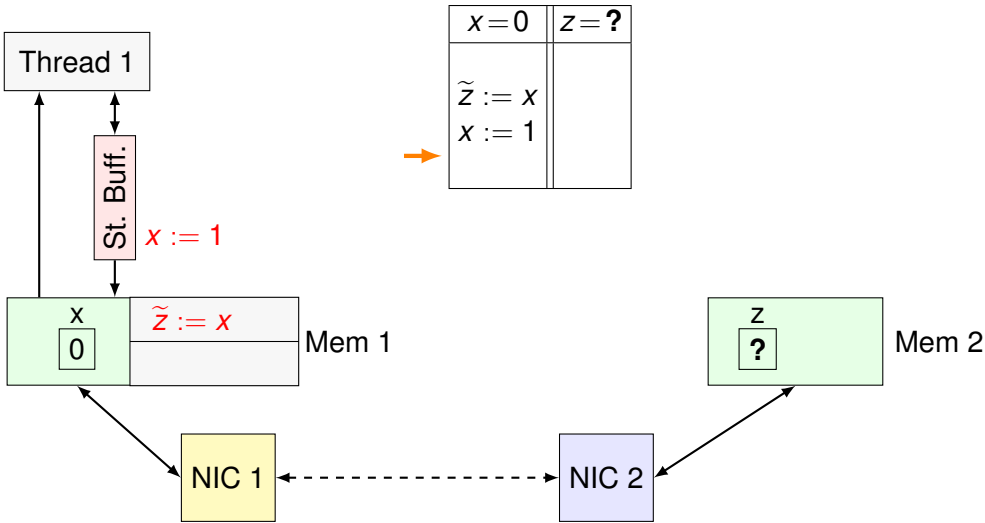
$x = 0$	$z = \mathbf{?}$
$\widetilde{z} := x$ $x := 1$	

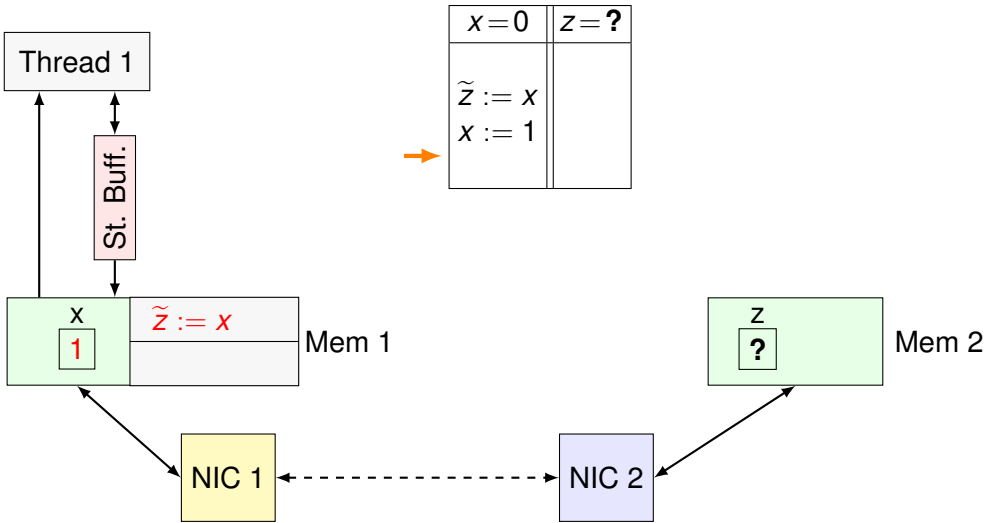
$z = \mathbf{?}$

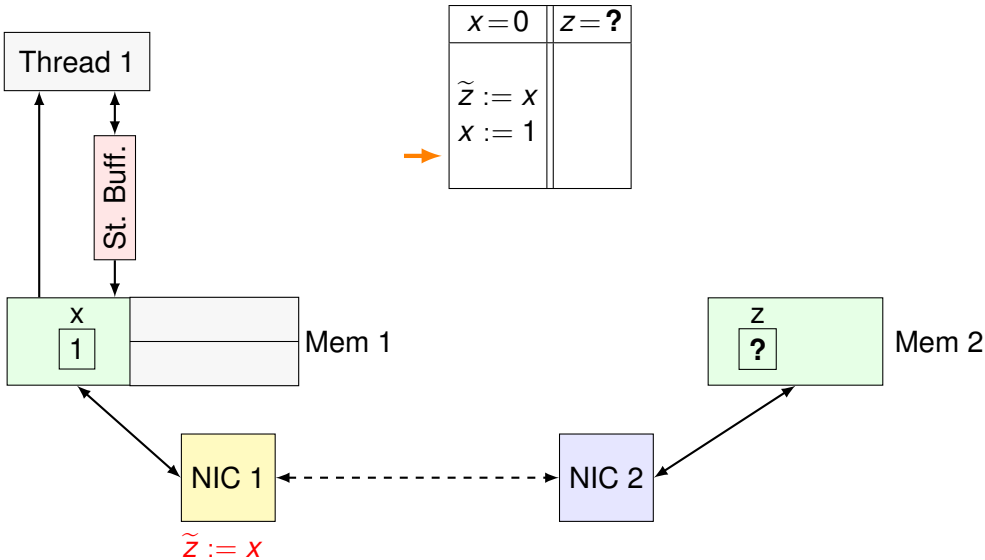

Thread 1
St. Buff.
x
0
Mem 1
z
?
Mem 2
NIC 1
NIC 2
$x = 0$
$z = ?$
$\widetilde{z} := x$
$x := 1$

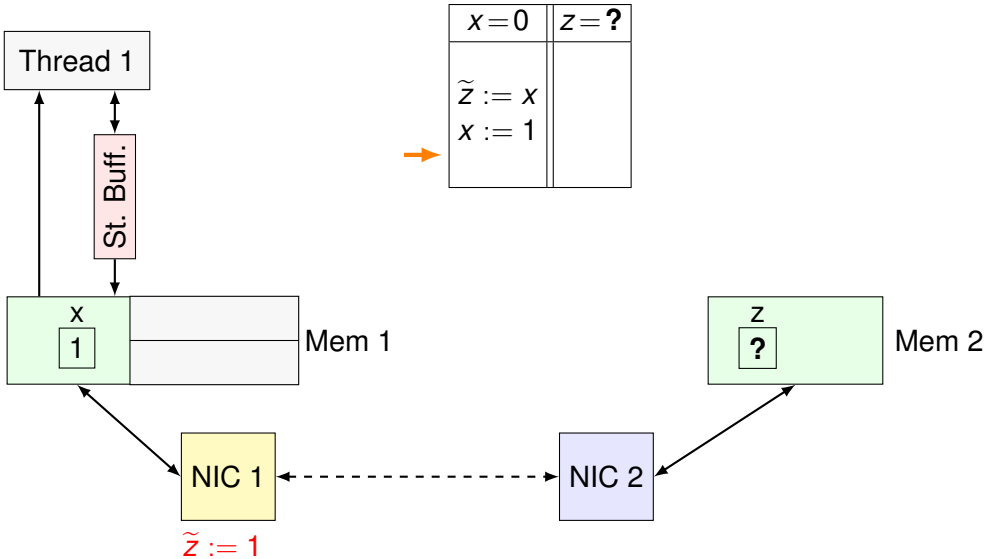

Thread 1
St. Buff.
$\widetilde{z} := x$
x
0
Mem 1
z
?
Mem 2
NIC 1
NIC 2
$x = 0$
$z = ?$
$\widetilde{z} := x$
$x := 1$

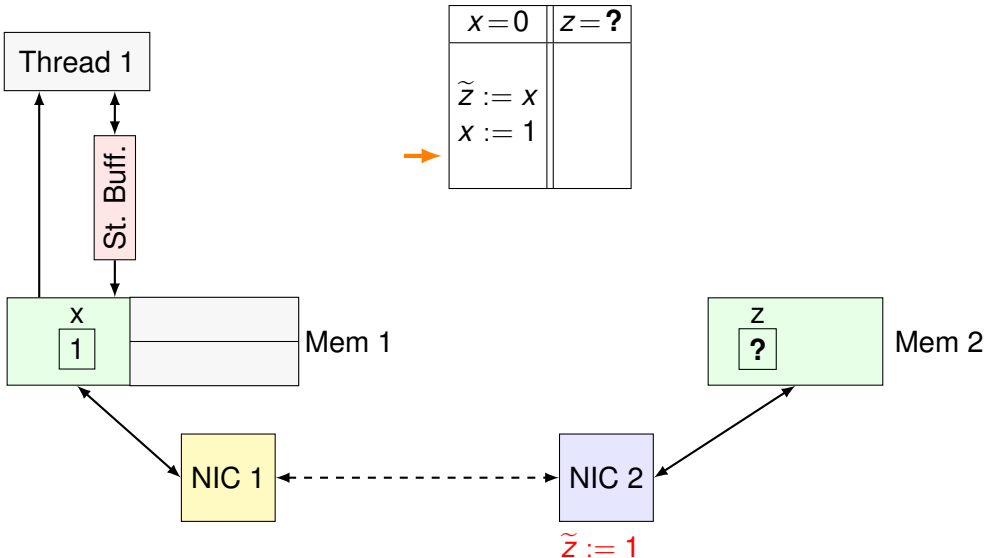

Thread 1
St. Buff.
x
0
$\widetilde{z} := x$
Mem 1
z
?
Mem 2
NIC 1
NIC 2
$x = 0$
$z = ?$
$\widetilde{z} := x$
$x := 1$

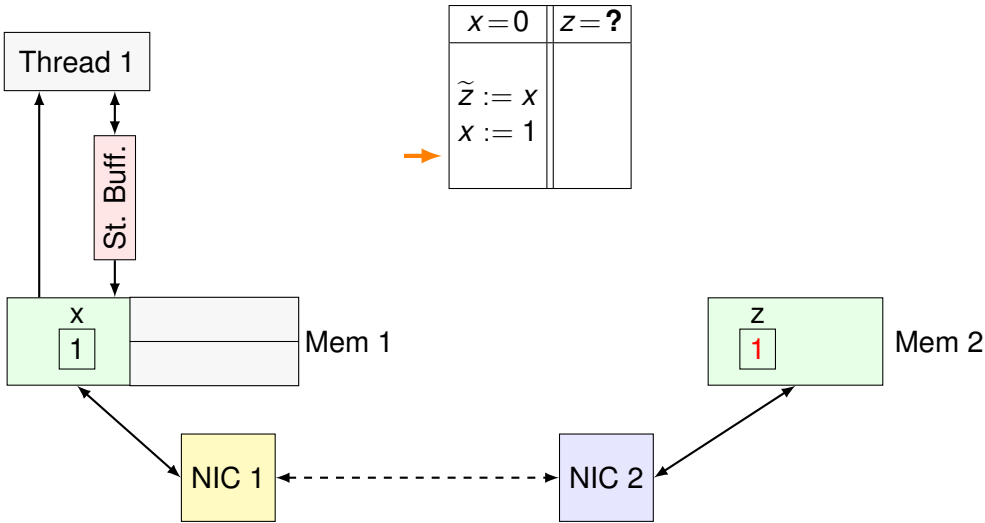

Thread 1
St. Buff.
$x := 1$
x
0
$\widetilde{z} := x$
Mem 1
$x = 0$
$z = ?$
$\widetilde{z} := x$
$x := 1$
z
?
Mem 2
NIC 1
NIC 2

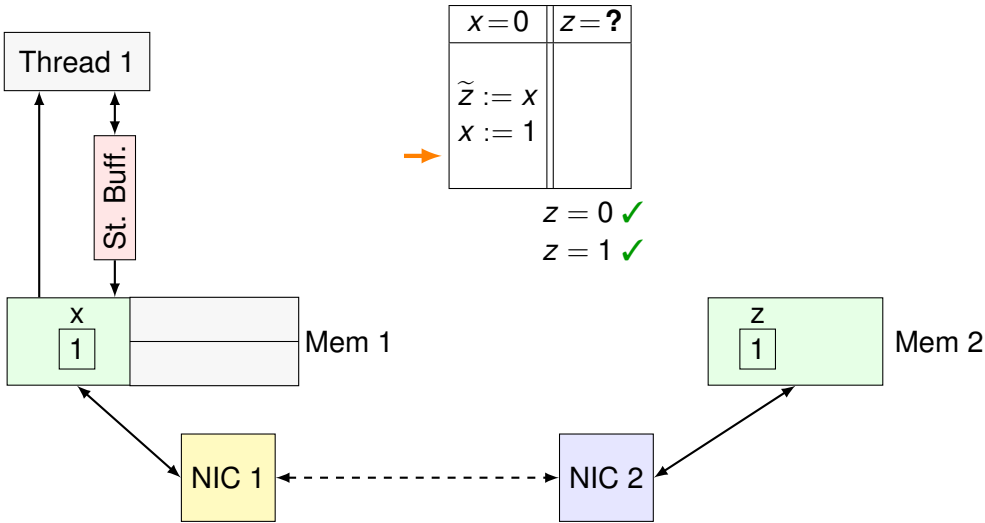

Thread 1
St. Buff.
x
1
$\widetilde{z} := x$
Mem 1
NIC 1
NIC 2
z
?
Mem 2
$x = 0$
$z = ?$
$\widetilde{z} := x$
$x := 1$


Thread 1
St. Buff.
x
1
Mem 1
z
?
Mem 2
NIC 1
NIC 2
$\widetilde{z} := x$
$x = 0$
$z = ?$
$\widetilde{z} := x$
$x := 1$


Thread 1
St. Buff.
x
1
Mem 1
z
?
Mem 2
NIC 1
NIC 2
$\widetilde{z} := 1$
$x = 0$
$z = ?$
$\widetilde{z} := x$
$x := 1$


Thread 1
St. Buff.
x
1
Mem 1
z
?
Mem 2
NIC 1
NIC 2
$x = 0$
$z = ?$
$\widetilde{z} := x$
$x := 1$
$\widetilde{z} := 1$

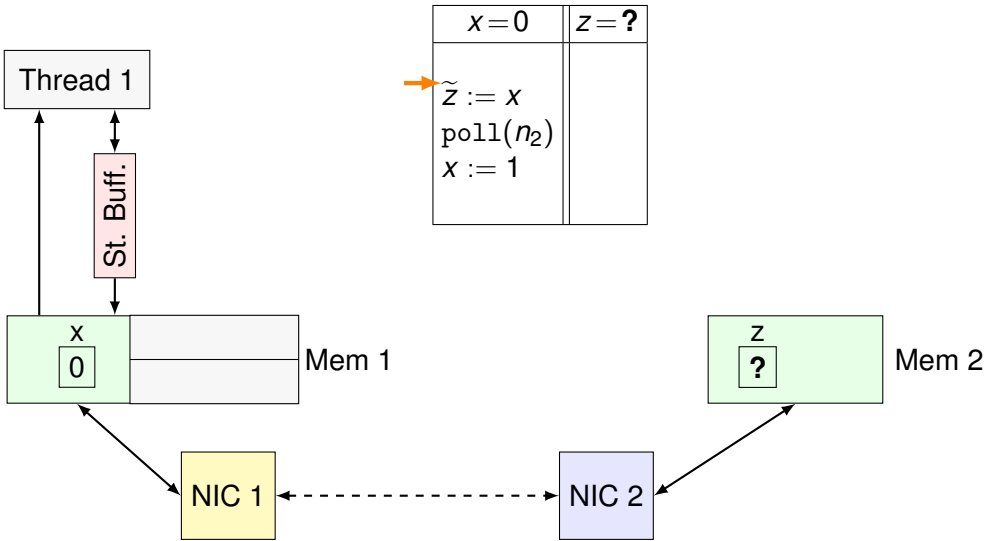

Thread 1
St. Buff.
x
1
Mem 1
z
1
Mem 2
NIC 1
NIC 2
$x = 0$
$z = ?$
$\widetilde{z} := x$
$x := 1$

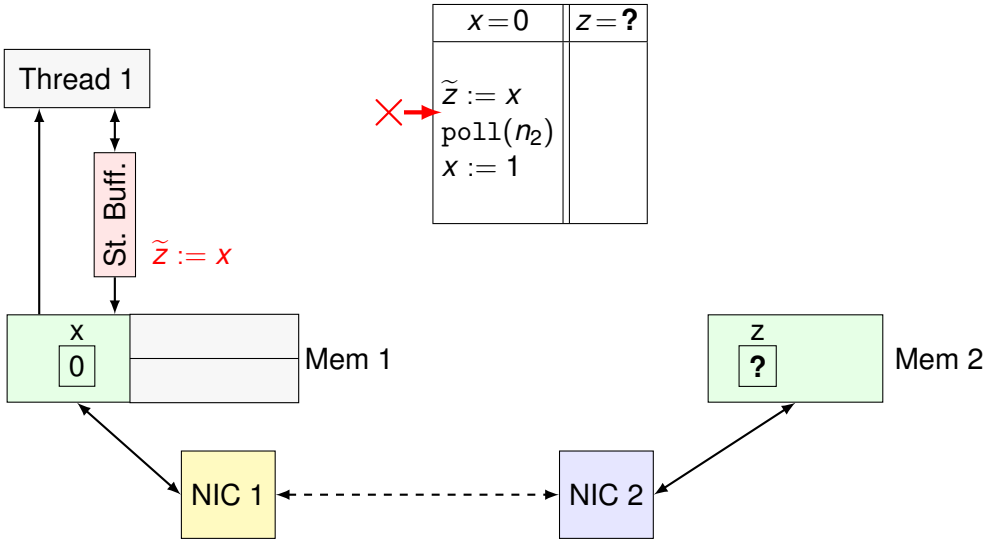

Thread 1
St. Buff.
x
1
Mem 1
z
1
Mem 2
NIC 1
NIC 2
$x = 0$
$z = ?$
$\widetilde{z} := x$
$x := 1$
$z = 0$ ✓
$z = 1$ ✓

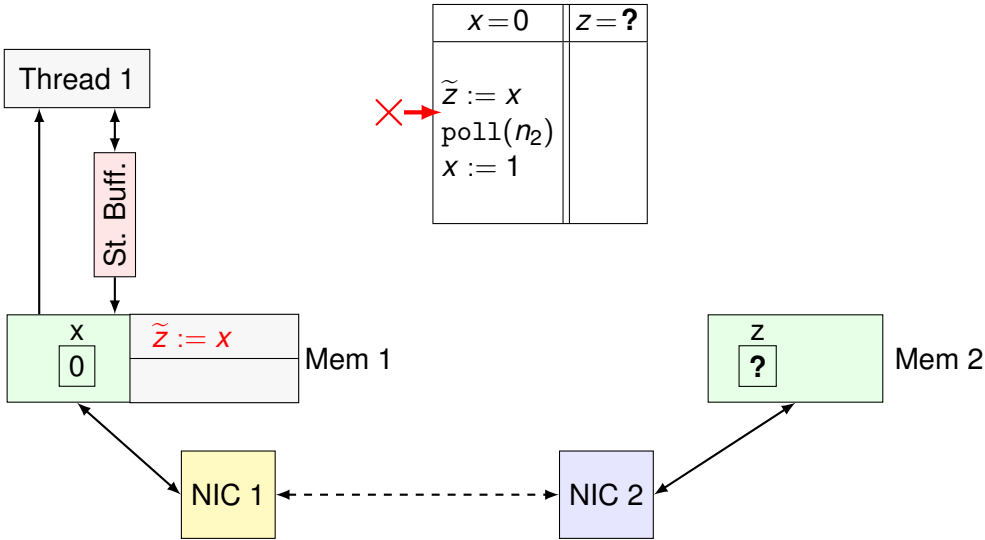
Example 3: Remote Write from Local – Poll – Local Write

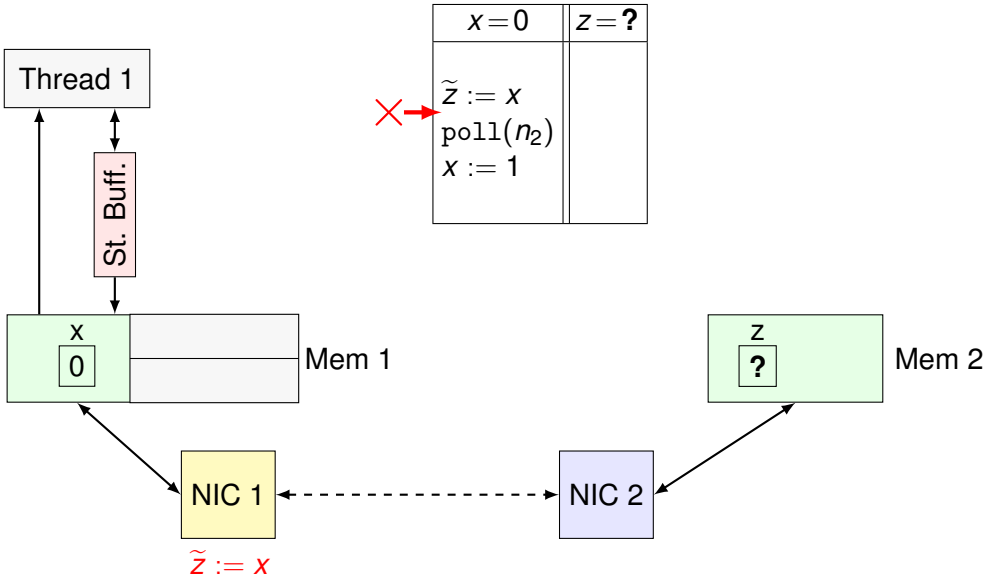
$x = 0$	$z = \mathbf{?}$
$\widetilde{z} := x$ $\texttt{poll}(n_2)$ $x := 1$	

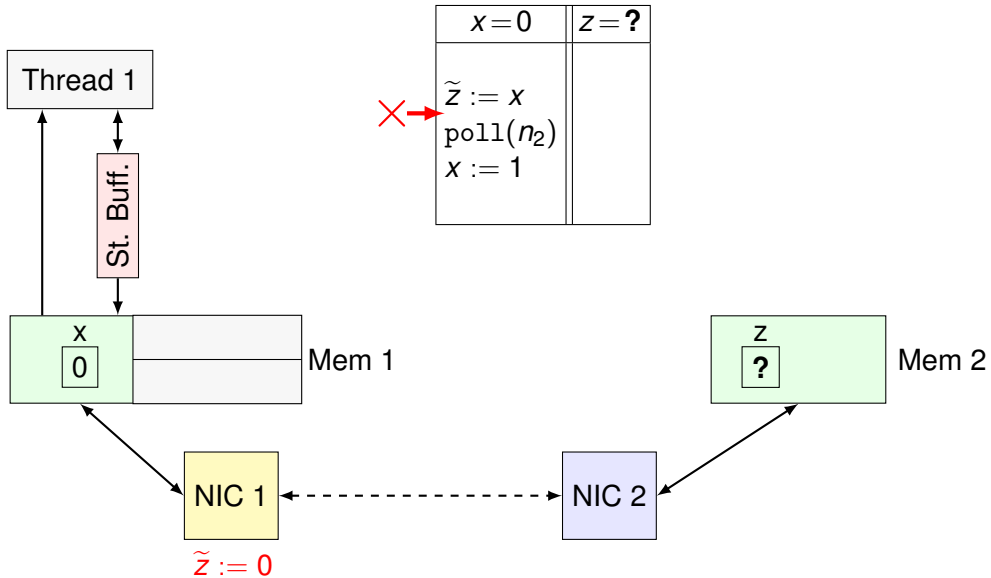
$z = \mathbf{?}$

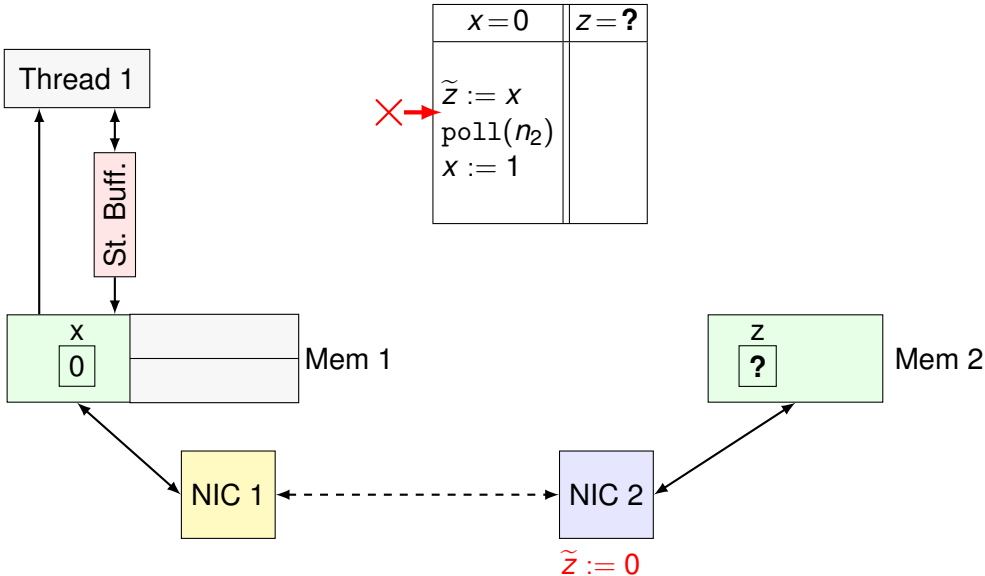

$x = 0$
$z = ?$
$\widetilde{z} := x$
poll(n_2)
$x := 1$
Thread 1
St. Buff.
x
0
Mem 1
z
?
Mem 2
NIC 1
NIC 2

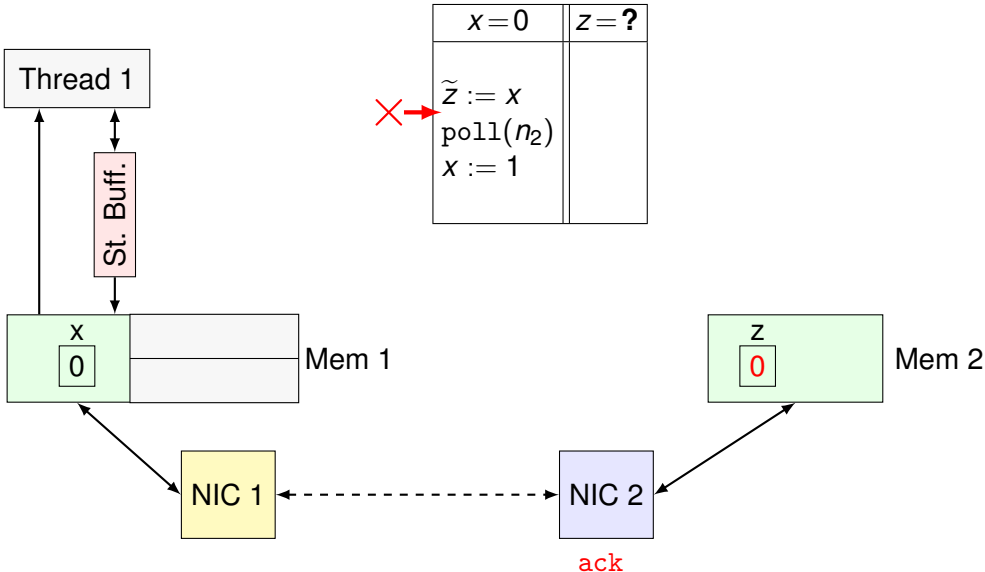

$x = 0$
$z = ?$
$\widetilde{z} := x$
poll(n_2)
$x := 1$
Thread 1
St. Buff.
$\widetilde{z} := x$
x
0
Mem 1
z
?
Mem 2
NIC 1
NIC 2

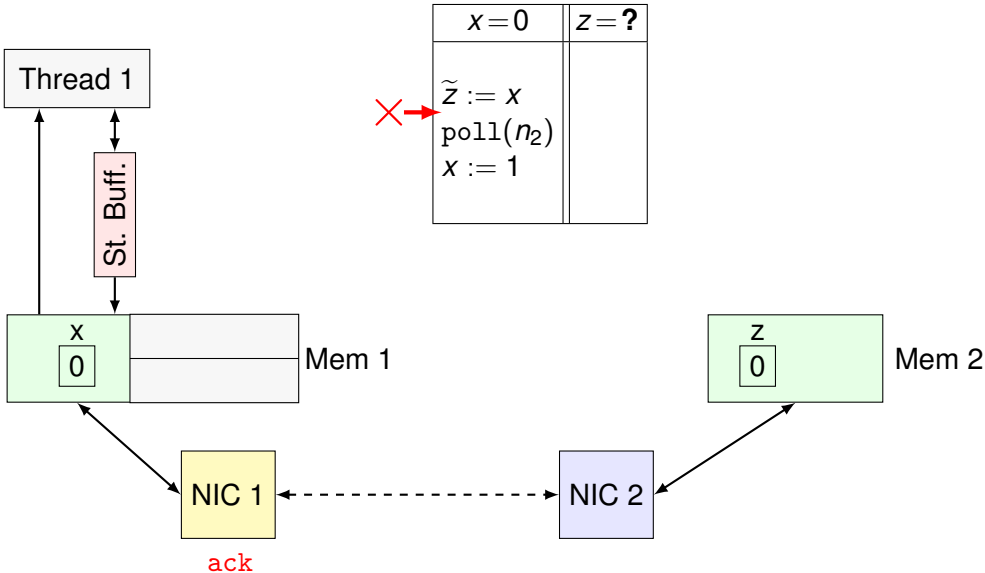

$x = 0$
$z = ?$
$\widetilde{z} := x$
poll(n_2)
$x := 1$
Thread 1
St. Buff.
x
0
$\widetilde{z} := x$
Mem 1
z
?
Mem 2
NIC 1
NIC 2

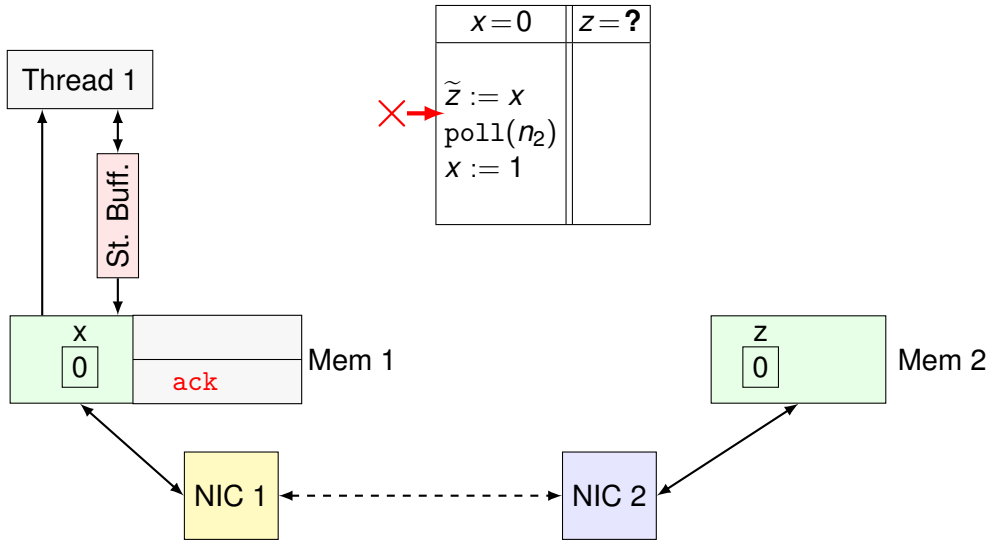

$x = 0$
$z = ?$
$\widetilde{z} := x$
poll(n_2)
$x := 1$
Thread 1
St. Buff.
x
0
Mem 1
z
?
Mem 2
NIC 1
NIC 2
$\widetilde{z} := x$

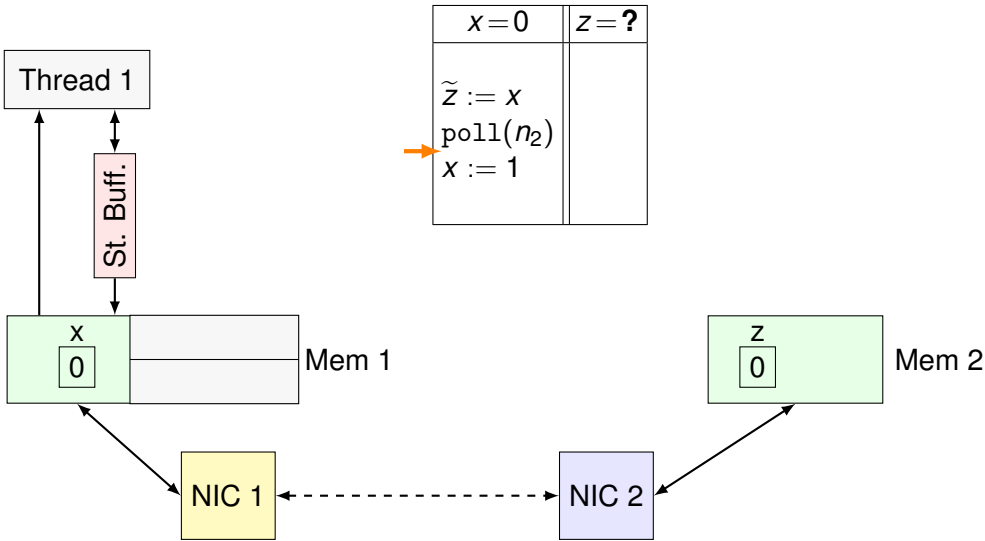

$x = 0$
$z = ?$
$\widetilde{z} := x$
poll(n_2)
$x := 1$
Thread 1
St. Buff.
x
0
Mem 1
z
?
Mem 2
NIC 1
NIC 2
$\widetilde{z} := 0$

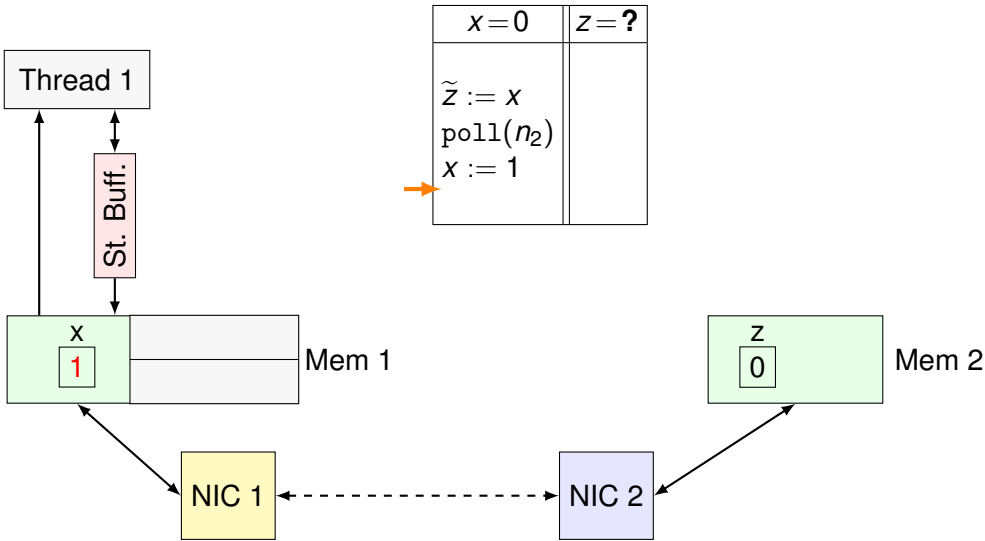

$x = 0$
$z = ?$
$\widetilde{z} := x$
poll(n_2)
$x := 1$
Thread 1
St. Buff.
x
0
Mem 1
z
?
Mem 2
NIC 1
NIC 2
$\widetilde{z} := 0$

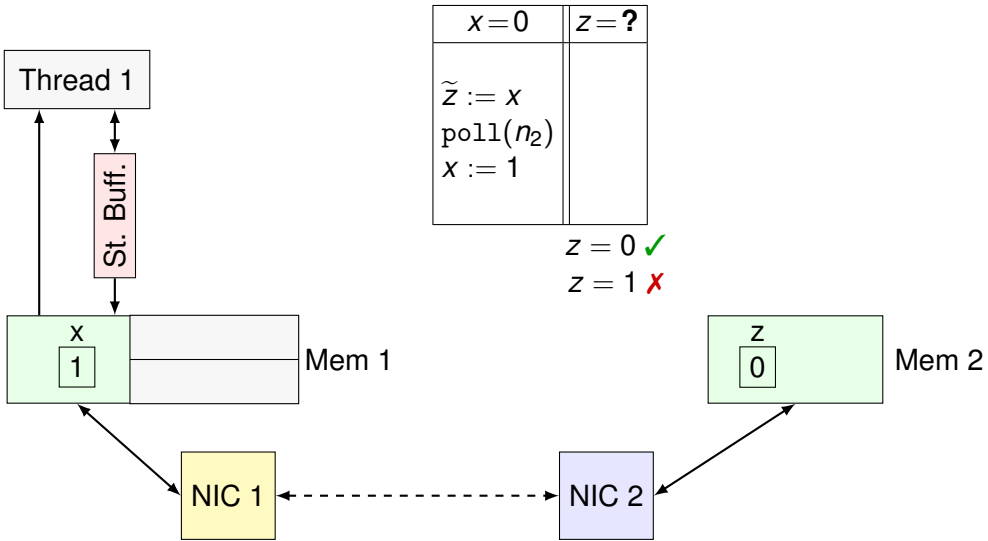

$x = 0$
$z = ?$
$\widetilde{z} := x$
poll(n_2)
$x := 1$
Thread 1
St. Buff.
x
0
Mem 1
z
0
Mem 2
NIC 1
NIC 2
ack


$x = 0$
$z = ?$
$\widetilde{z} := x$
poll(n_2)
$x := 1$
Thread 1
St. Buff.
x
0
Mem 1
z
0
Mem 2
NIC 1
NIC 2
ack


$x = 0$
$z = ?$
$\widetilde{z} := x$
poll(n_2)
$x := 1$
Thread 1
St. Buff.
x
0
ack
Mem 1
z
0
Mem 2
NIC 1
NIC 2


$x = 0$
$z = ?$
$\widetilde{z} := x$
poll(n_2)
$x := 1$
Thread 1
St. Buff.
x
0
Mem 1
z
0
Mem 2
NIC 1
NIC 2


$x = 0$
$z = ?$
$\widetilde{z} := x$
poll(n_2)
$x := 1$
Thread 1
St. Buff.
x
1
Mem 1
z
0
Mem 2
NIC 1
NIC 2


$x = 0$
$z = ?$
$\widetilde{z} := x$
poll(n_2)
$x := 1$
$z = 0$ ✓
$z = 1$ ✗
Thread 1
St. Buff.
x
1
Mem 1
z
0
Mem 2
NIC 1
NIC 2

Problems with RDMA^{TSO}

Problem 1: Message passing needs remote fences

$x, y = 0$	
$x := 1$ $y := 1$	$a := \widetilde{y}$ $b := \widetilde{x}$

$(a, b) = (1, 0)$ ✓

Problem 1: Message passing needs remote fences

$x, y = 0$	
$x := 1$ $y := 1$	$a := \widetilde{y}$ $b := \widetilde{x}$

$(a, b) = (1, 0)$ ✓

$x, y = 0$	
$x := 1$ $y := 1$	$a := \widetilde{y}$ `rfence` (n_1) $b := \widetilde{x}$

$(a, b) = (1, 0)$ ✗

Problem 2: Remote operations are highly asynchronous

$x=0$	$z=1$	$y=2$
$x := \widetilde{z}$ $x := \widetilde{y}$		

$x=1$ ✓

$x=2$ ✓

Problem 2: Remote operations are highly asynchronous

$x=0$	$z=1$	$y=2$
$x := \widetilde{z}$ $x := \widetilde{y}$		

$x=1$ ✓
$x=2$ ✓

$x=0$	$z=1$	$y=2$
$\widetilde{y} := x$ $x := \widetilde{z}$		

$y=0$ ✓
$y=1$ ✓

Problem 3: Poll-based synchronisation is very fragile

$x = 0$	$z = 0$
$\widetilde{z} := x$ $\texttt{poll}(n_2)$ $x := 1$	

$z = 0$ ✓

$z = 1$ ✗

Problem 3: Poll-based synchronisation is very fragile

$x=0$	$z=0$
$\widetilde{z} := x$ $\texttt{poll}(n_2)$ $x := 1$	

$z=0$ ✓
$z=1$ ✗

$x=0$	$z=0$
$\widetilde{z} := x$ $\widetilde{z} := x$ $\texttt{poll}(n_2)$ $x := 1$	

$z=0$ ✓
$z=1$ ✓

Problem 3: Poll-based synchronisation is very fragile

$x=0$	$z=0$
$\widetilde{z} := x$ $\texttt{poll}(n_2)$ $x := 1$	

$z=0$ ✓
$z=1$ ✗

$x=0$	$z=0$
$\widetilde{z} := x$ $\widetilde{z} := x$ $\texttt{poll}(n_2)$ $x := 1$	

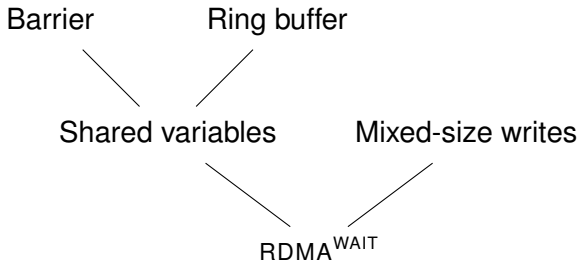
$z=0$ ✓
$z=1$ ✓

$x=0$	$z=0$
$\widetilde{z} := x$ $\widetilde{z} := x$ $\texttt{poll}(n_2)$ $\texttt{poll}(n_2)$ $x := 1$	

$z=0$ ✓
$z=1$ ✗

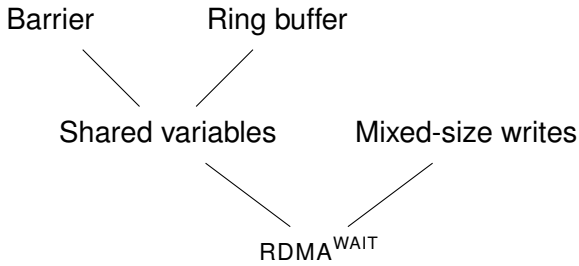
LOCO Libraries

LOCO abstractions



- LOCO (Library of Channel Objects) is a tower of abstractions emulating shared memory under RDMA (`https://arxiv.org/abs/2503.19270`)
- New base memory model RDMA$^{\text{WAIT}}$
- Highly performant and composable

LOCO abstractions



- LOCO (Library of Channel Objects) is a tower of abstractions emulating shared memory under RDMA (`https://arxiv.org/abs/2503.19270`)
- New base memory model RDMA$^{\text{WAIT}}$
- Highly performant and composable
- ... BUT... bugs found!

LOCO: RDMA$^{\text{WAIT}}$

Recall:

$x=0$	$z=0$
$\widetilde{z} := x$ $\widetilde{z} := x$ $\texttt{poll}(n_2)$ $x := 1$	

$z=0$ ✓
$z=1$ ✓

Recall:

$x=0$	$z=0$
$\widetilde{z} := x$ $\widetilde{z} := x$ $\texttt{poll}(n_2)$ $x := 1$	

$z=0$ ✓
$z=1$ ✓

`wait` behaviour:

$x=0$	$z=0$
$\widetilde{z} :=^{d} x$ $\texttt{wait}(d)$ $x := 1$	

$z=0$ ✓
$z=1$ ✗

Recall:

$x=0$	$z=0$
$\widetilde{z} := x$ $\widetilde{z} := x$ $\texttt{poll}(n_2)$ $x := 1$	

$z=0$ ✓
$z=1$ ✓

`wait` behaviour:

$x=0$	$z=0$
$\widetilde{z} :=^{d} x$ $\texttt{wait}(d)$ $x := 1$	

$z=0$ ✓
$z=1$ ✗

$x=0$	$z=0$
$\widetilde{z} :=^{e} x$ $\widetilde{z} :=^{d} x$ $\texttt{wait}(d)$ $x := 1$	

$z=0$ ✓
$z=1$ ✗

Recall:

$x=0$	$z=0$
$\widetilde{z} := x$ $\widetilde{z} := x$ $\texttt{poll}(n_2)$ $x := 1$	

$z=0$ ✓
$z=1$ ✓

`wait` **behaviour:**

$x=0$	$z=0$
$\widetilde{z} :=^{d} x$ $\texttt{wait}(d)$ $x := 1$	

$z=0$ ✓
$z=1$ ✗

$x=0$	$z=0$
$\widetilde{z} :=^{e} x$ $\widetilde{z} :=^{d} x$ $\texttt{wait}(d)$ $x := 1$	

$z=0$ ✓
$z=1$ ✗

RDMA$^{\text{WAIT}}$ is implemented over RDMA$^{\text{TSO}}$

Wait and Store Buffering

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

$(a, b) = (0, 0)$ **?**

Wait and Store Buffering

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

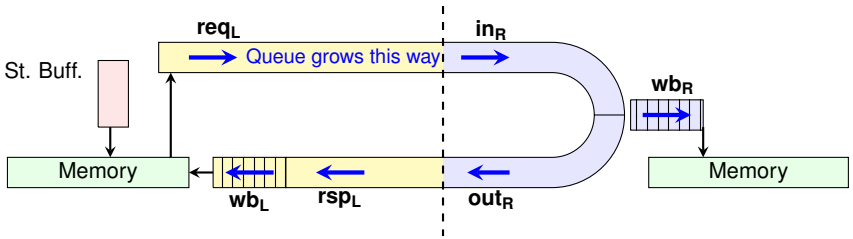
$(a, b) = (0, 0)$ ✓

Wait and Store Buffering

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

$(a, b) = (0, 0)$ ✓

`wait(_)` only guarantees that the write has reached the remote buffer



Fixing Store Buffering with Wait

$y = 0$	$x = 0$
$\widetilde{x} := 1$	$\widetilde{y} := 1$
$\bot :=^{d} \bot^{n_2}$	$\bot :=^{e} \bot^{n_1}$
$\texttt{wait}(d)$	$\texttt{wait}(e)$
$a := y$	$b := x$

$(a, b) = (0, 0)$ ✗

- $\bot :=^{d} \bot$ is a zero-length read

Fixing Store Buffering with Wait

$y = 0$	$x = 0$
$\widetilde{x} := 1$ $\bot :=^{d} \bot^{n_2}$ $\texttt{wait}(d)$ $a := y$	$\widetilde{y} := 1$ $\bot :=^{e} \bot^{n_1}$ $\texttt{wait}(e)$ $b := x$

- $\bot :=^{d} \bot$ is a zero-length read
- But what if we want to
  - synchronise across k nodes, or
  - write to multiple nodes?

LOCO: Shared Variables

Shared Variables

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

Idea: A copy of x exists on all nodes

Shared Variables

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

Idea: A copy of x exists on all nodes

- $\texttt{Write}_{\texttt{SV}}(x, v)$: modify x locally
- $\texttt{Read}_{\texttt{SV}}(x, a)$: return the local value of x into a

Shared Variables

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

Idea: A copy of x exists on all nodes

- $\texttt{Write}_{\texttt{SV}}(x, v)$: modify x locally
- $\texttt{Read}_{\texttt{SV}}(x, a)$: return the local value of x into a
- $\texttt{GF}_{\texttt{SV}}(\{n_1, \ldots, n_k\})$: perform a global fence on given set of nodes

Shared Variables

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

Idea: A copy of x exists on all nodes

- $\texttt{Write}_{\texttt{SV}}(x, v)$: modify x locally
- $\texttt{Read}_{\texttt{SV}}(x, a)$: return the local value of x into a
- $\texttt{GF}_{\texttt{SV}}(\{n_1, \ldots, n_k\})$: perform a global fence on given set of nodes
- $\texttt{Bcast}_{\texttt{SV}}(x, d, \{n_1, \ldots, n_k\})$: broadcast x with to nodes with broadcast id d
- $\texttt{Wait}_{\texttt{SV}}(d)$: wait for broadcast with id d

Shared Variables

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

Idea: A copy of x exists on all nodes

- $\texttt{Write}_{\texttt{SV}}(x, v)$: modify x locally
- $\texttt{Read}_{\texttt{SV}}(x, a)$: return the local value of x into a
- $\texttt{GF}_{\texttt{SV}}(\{n_1, \ldots, n_k\})$: perform a global fence on given set of nodes
- $\texttt{Bcast}_{\texttt{SV}}(x, d, \{n_1, \ldots, n_k\})$: broadcast x with to nodes with broadcast id d
- $\texttt{Wait}_{\texttt{SV}}(d)$: wait for broadcast with id d

Note: SV allows write-write races; prevented using another library called *Owned Variables*

Global Fence Behaviour

Recall:

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

$(a, b) = (0, 0)$ ✓

$y=0$	$x=0$
$\widetilde{x} := 1$	$\widetilde{y} :=^{e} 1$
$\perp :=^{d} \perp^{n_2}$	$\perp :=^{e} \perp^{n_1}$
$\texttt{wait}(d)$	$\texttt{wait}(e)$
$a := y$	$b := x$

$(a, b) = (0, 0)$ ✗

Global Fence Behaviour

Recall:

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

$(a, b) = (0, 0)$ ✓

$y=0$	$x=0$
$\widetilde{x} := 1$	$\widetilde{y} :=^{e} 1$
$\bot :=^{d} \bot^{n_2}$	$\bot :=^{e} \bot^{n_1}$
$\texttt{wait}(d)$	$\texttt{wait}(e)$
$a := y$	$b := x$

$(a, b) = (0, 0)$ ✗

Global Fence:

$y = 0$	$x = 0$
$\widetilde{x} := 1$	$\widetilde{y} := 1$
$\texttt{GF}_{\texttt{SV}}(\{n_2\})$	$\texttt{GF}_{\texttt{SV}}(\{n_1\})$
$a := y$	$b := x$

$(a, b) = (0, 0)$ ✗

LOCO: Shared Variables Implementation

Shared Variables: Implementation

Shared variable library is implemented over $\text{RDMA}^{\text{WAIT}}$

Shared Variables: Implementation

Shared variable library is implemented over $\text{RDMA}^{\text{WAIT}}$

$$\begin{aligned} \texttt{Write}_{\text{SV}}(x, v) &\rightsquigarrow x := v \\ \texttt{Read}_{\text{SV}}(x, a) &\rightsquigarrow a := x \end{aligned}$$

Shared Variables: Implementation

Shared variable library is implemented over $\text{RDMA}^{\text{WAIT}}$

$$
\begin{aligned}
\texttt{Write}_{\text{SV}}(x, v) &\rightsquigarrow x := v \\
\texttt{Read}_{\text{SV}}(x, a) &\rightsquigarrow a := x \\
\texttt{GF}_{\text{SV}}(\{n_1, \ldots, n_k\}) &\rightsquigarrow \bot :=^{d} \bot^{n_1} ; \ldots ; \bot :=^{d} \bot^{n_k} ; \texttt{wait}(d)
\end{aligned}
$$

Shared Variables: Implementation

Shared variable library is implemented over $\text{RDMA}^{\text{WAIT}}$

$$
\begin{aligned}
\texttt{Write}_{\text{SV}}(x, v) &\rightsquigarrow x := v \\
\texttt{Read}_{\text{SV}}(x, a) &\rightsquigarrow a := x \\
\texttt{GF}_{\text{SV}}(\{n_1, \ldots, n_k\}) &\rightsquigarrow \bot :=^{d} \bot^{n_1} ; \ldots ; \bot :=^{d} \bot^{n_k} ; \texttt{wait}(d) \\
\texttt{Bcast}_{\text{SV}}(x, d, \{n_1, \ldots, n_k\}) &\rightsquigarrow x^{n_1} :=^{d} x; \ldots ; x^{n_k} :=^{d} x \\
\texttt{Wait}_{\text{SV}}(d) &\rightsquigarrow \texttt{wait}(d)
\end{aligned}
$$

Shared Variables: Implementation

Shared variable library is implemented over $\text{RDMA}^{\text{WAIT}}$

$$
\begin{aligned}
\texttt{Write}_{\text{SV}}(x, v) &\rightsquigarrow x := v \\
\texttt{Read}_{\text{SV}}(x, a) &\rightsquigarrow a := x \\
\texttt{GF}_{\text{SV}}(\{n_1, \ldots, n_k\}) &\rightsquigarrow \bot :=^{d} \bot^{n_1} ; \ldots ; \bot :=^{d} \bot^{n_k} ; \texttt{wait}(d) \\
\texttt{Bcast}_{\text{SV}}(x, d, \{n_1, \ldots, n_k\}) &\rightsquigarrow x^{n_1} :=^{d} x; \ldots; x^{n_k} :=^{d} x \\
\texttt{Wait}_{\text{SV}}(d) &\rightsquigarrow \texttt{wait}(d)
\end{aligned}
$$

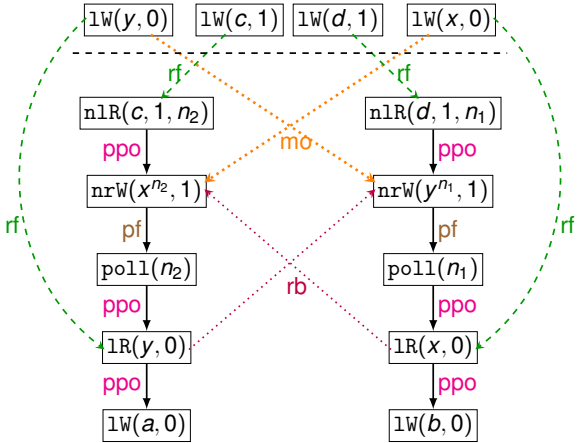
Question: How do we show correctness of the implementation?

Declarative Verification Framework

Declarative Semantics

$y, c = 0, 1$	$x, d = 0, 1$
$\widetilde{x} := c$ $\texttt{poll}(n_2)$ $a := y$	$\widetilde{y} := d$ $\texttt{poll}(n_1)$ $b := x$

$(a, b) = (0, 0)$ ✓



Declarative Correctness

Use a *declarative* (aka axiomatic) style of semantics

1. Define *consistency predicate* for the specification
2. Prove relationship between implementation and specification via an abstraction function

Declarative Correctness

Use a *declarative* (aka axiomatic) style of semantics

1. Define *consistency predicate* for the specification
2. Prove relationship between implementation and specification via an abstraction function

.... BUT .... there are many details:

- Compositionality: Consistency for a program using a *set* of libraries
- Subevents: Need relations at a finer granularity than those over atomic actions of a specification
- Preserved program order (ppo): Program order is too strong to define global consistency
- Specification order (so): Needed to define happens-before guarantees of each library

Subevents via Stamps

Use “stamps” to split events into fine-grained subevents

$$\text{SEvent} \triangleq \{\langle \mathsf{e}, a \rangle \mid \mathsf{e} \in E \wedge a \in \texttt{stmp}(\mathsf{e})\}$$

Subevents via Stamps

Use “stamps” to split events into fine-grained subevents

$$\text{SEvent} \triangleq \{\langle \mathsf{e}, a\rangle \mid \mathsf{e} \in E \wedge a \in \mathtt{stmp}(\mathsf{e})\}$$

Example. For the shared variable library:

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

Stamp Order and Preserved Program Order

Stamps order (to) used to define *preserved program order* (ppo)

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

Stamp Order and Preserved Program Order

Stamps order (to) used to define *preserved program order* (ppo)

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

Example. Shared variable library

Second Stamp

First Stamp			to	single					families					
				1	2	3	4	5	6	7	8	9	10	11
				aCR	aCW	aCAS	aMF	aWT	aNLR$_n$	aNRW$_n$	aNRR$_n$	aNLW$_n$	aRF$_n$	aGF$_n$
single	A	aCR		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	B	aCW		✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
	C	aCAS		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	D	aMF		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	E	aWT		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
families	F	aNLR$_n$		✗	✗	✗	✗	✗	SN	SN	SN	SN	SN	SN
	G	aNRW$_n$		✗	✗	✗	✗	✗	✗	SN	SN	SN	✗	SN
	H	aNRR$_n$		✗	✗	✗	✗	✗	✗	✗	✗	SN	SN	SN
	I	aNLW$_n$		✗	✗	✗	✗	✗	✗	✗	✗	SN	✗	SN
	J	aRF$_n$		✗	✗	✗	✗	✗	SN	SN	SN	SN	SN	SN
	K	aGF$_n$		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Global Consistency

Definition

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

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

Per Library Consistency

Define the specification of each library using notion of consistency

Definition (SV-consistency)

$\langle E, \mathsf{po}, \mathtt{stmp}, \mathsf{so}, \mathsf{hb} \rangle$ is SV-consistent if:

1. there exists rf, and mo, such that
   1. $[\mathtt{aCR}]; (\mathsf{po}^{-1} \cap \mathsf{rb}); [\mathtt{aCW}] = \emptyset$, and
   2. $\mathsf{so} = \mathsf{iso} \cup \mathsf{rf_e} \cup \mathsf{pf} \cup \mathsf{rb} \cup \mathsf{mo}$.

iso, $\mathsf{rf_e}$, pf, rb and mo are further relations derived from po, rf and mo.

Implementation Soundness and Locality

- **Global soundness** (aka contextual refinement):
  For every output of a program using a (family of) library implementation(s)

Implementation Soundness and Locality

- **Global soundness** (aka contextual refinement):

  For every output of a program using a (family of) library implementation(s)
  there exists a valid execution of the program using a (family of) specification(s)
  producing the same output

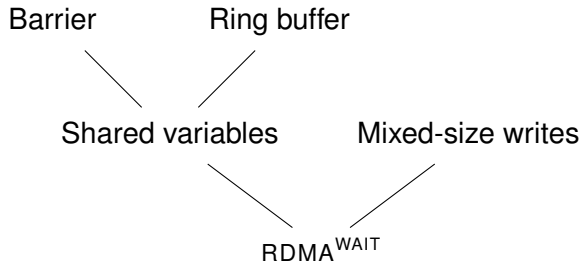
Implementation Soundness and Locality

- **Global soundness** (aka contextual refinement):
  For every output of a program using a (family of) library implementation(s)
  there exists a valid execution of the program using a (family of) specification(s)
  producing the same output

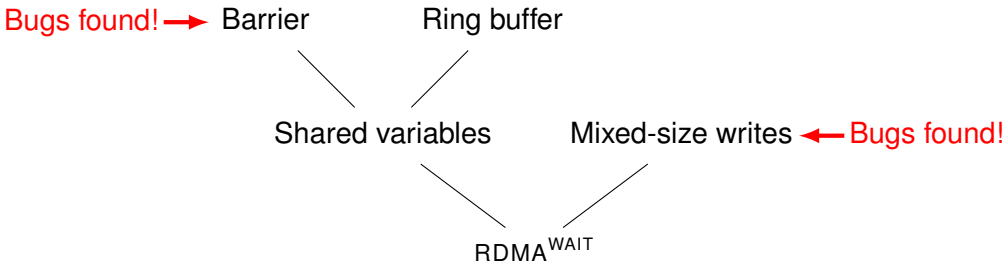
- **Local soundness**:
  Conditions for relating execution graphs of a library's implementation and specification
  (straightforward, but technical)

- **Locality Theorem**:
  If an implementation is locally sound, then it is globally sound.

Summary of Proofs



Summary of Proofs



Conclusions

- Technologies such as RDMA (and CXL) are becoming increasingly important for datacenters, cloud servers, distributed AI training / federated ML, etc
- Libraries such as LOCO provide programmer-friendly high-performance abstractions
- We are taking steps towards correctness

Conclusions

- Technologies such as RDMA (and CXL) are becoming increasingly important for datacenters, cloud servers, distributed AI training / federated ML, etc
- Libraries such as LOCO provide programmer-friendly high-performance abstractions
- We are taking steps towards correctness ... BUT ... we have many open questions

Conclusions

- Technologies such as RDMA (and CXL) are becoming increasingly important for datacenters, cloud servers, distributed AI training / federated ML, etc
- Libraries such as LOCO provide programmer-friendly high-performance abstractions
- We are taking steps towards correctness ... BUT ... we have many open questions
- Future work:
  - Scalable (operational) proofs
  - (Owicki/Gries) logics
  - Mechanisation
  - ...

Conclusions

- Technologies such as RDMA (and CXL) are becoming increasingly important for datacenters, cloud servers, distributed AI training / federated ML, etc
- Libraries such as LOCO provide programmer-friendly high-performance abstractions
- We are taking steps towards correctness ... BUT ... we have many open questions
- Future work:
  - Scalable (operational) proofs
  - (Owicki/Gries) logics
  - Mechanisation
  - ...
- Dagstuhl seminar proposal on “Foundations of Disaggregated Memory and Heterogeneous Architectures” (under review)