A Verified High-Performance Composable Object Library for Remote Direct Memory Access

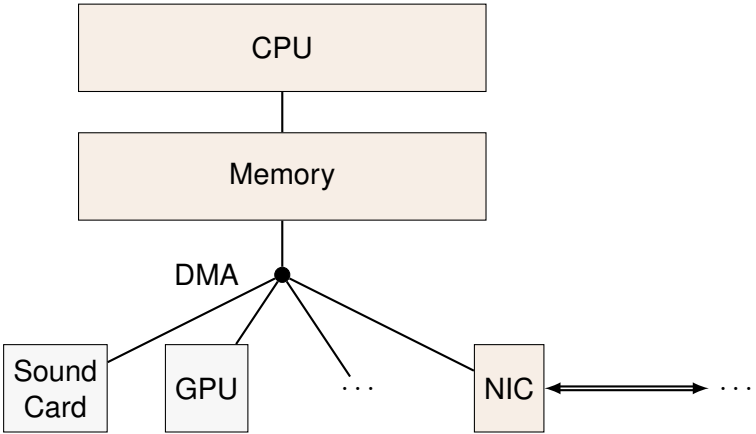
POPL 2026

Guillaume Ambal[1], George Hodgkins[2], Mark Madler[2], Gregory Chockler[3], Brijesh Dongol[3], Joseph Izraelevitz[2], Azalea Raad[1], Viktor Vafeiadis[4]

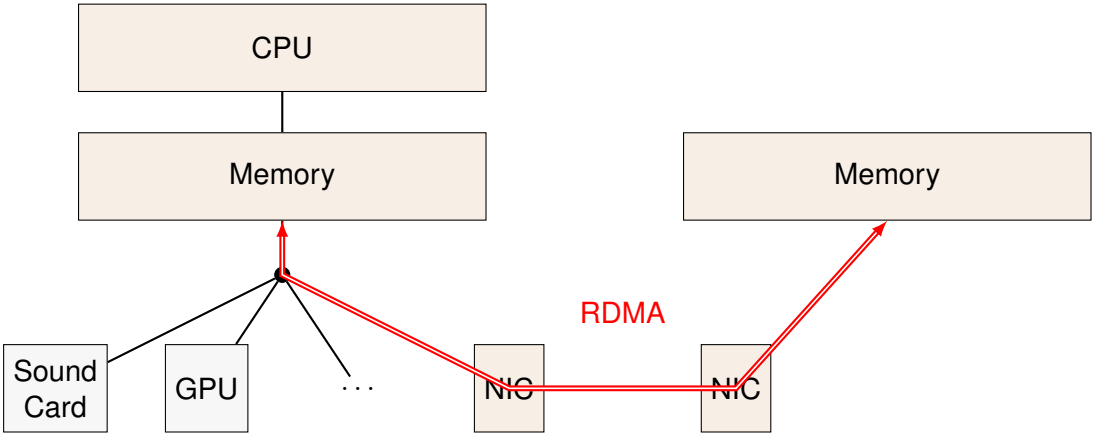
[1]Imperial College London, [2]University of Colorado, Boulder, [3]University of Surrey, [4]MPI-SWS

January 16, 2026

Remote Direct Memory Access (RDMA)



Remote Direct Memory Access (RDMA)



RDMA: Data-transfer protocol for High-Performance Computing
Low Latency: $\sim \mu$s

RDMA^{TSO} [OOPSLA 24]

Two nodes, one thread

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

$z = 0$ ✓ $z = 1$**?**

Can the output be $z = 1$?

RDMA^{TSO} [OOPSLA 24]

Two nodes, one thread

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

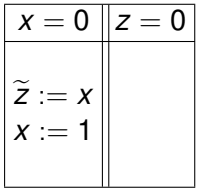
$z = 0$ ✓ $z = 1$ ✓

Can the output be $z = 1$?

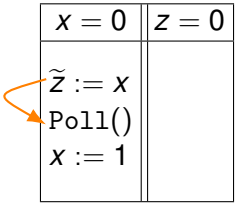
Yes! like this:

- CPU offloads "$\widetilde{z} := x$" to the NIC
- CPU executes "$x := 1$"
- NIC picks up "$\widetilde{z} := x$" and reads 1
- ...

RDMA^{TSO} [OOPSLA 24]



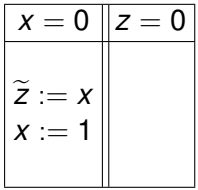
$z = 1$ ✓



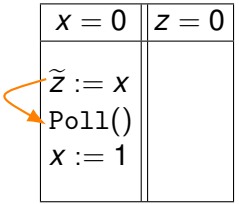
$z = 1$ ✗

Can prevent this!

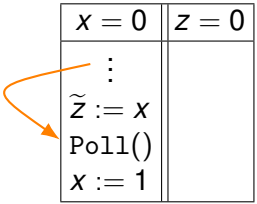
RDMA^{TSO} [OOPSLA 24]



$z = 1$ ✓



$z = 1$ ✗



$z = 1$ ✓

Can prevent this!

But it's context-dependent...

Problems and Contributions

Problems with current RDMA programs:

- Low-level non-modular code (no libraries)
- No formalisation or verification

Problems and Contributions

Problems with current RDMA programs:

- Low-level non-modular code (no libraries)
- No formalisation or verification

Our contributions:

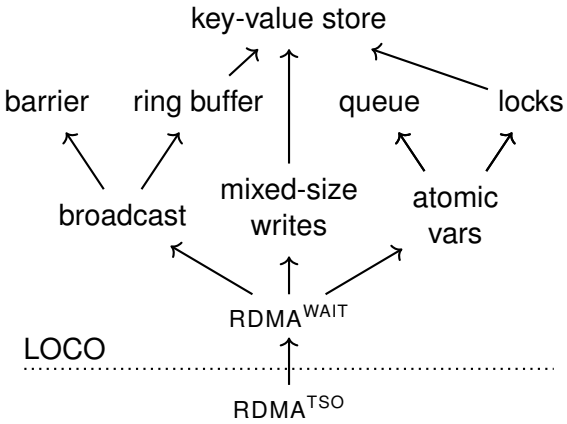
- Generalised RDMA^{TSO} to $\text{RDMA}^{\text{WAIT}}$ as a modular alternative
- LOCO: High-performance Modular RDMA Libraries
- Mowgli: Framework for Libraries in (very) weak settings such as RDMA
- Local soundness result for simpler verification
- Specified and verified a large fragment of LOCO using Mowgli

LOCO: Overview

LOCO (Library of Composable Objects)

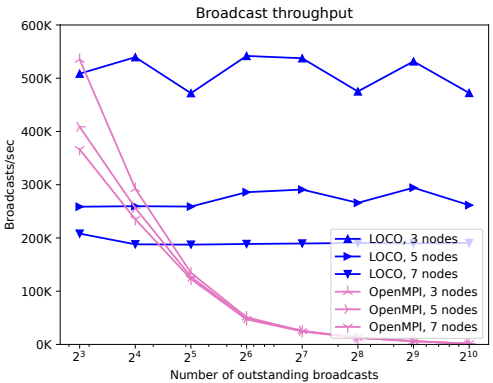
One manager thread talks to the hardware
$\Rightarrow$ RDMA$^{\text{WAIT}}$ (context independent code)

Tower of libraries, allowing programming RDMA similarly to shared-memory

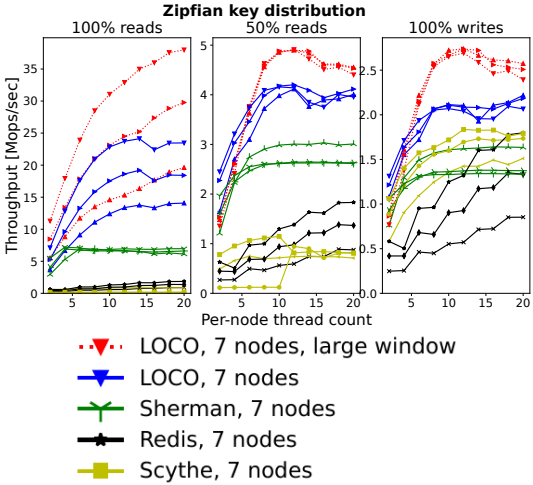


LOCO: High-Performance Libraries

Broadcast Benchmark



Key-Values Store Benchmark



Specification: Ring Buffer Library

Example 1: Ring Buffer (single-writer-multiple-reader, FIFO)

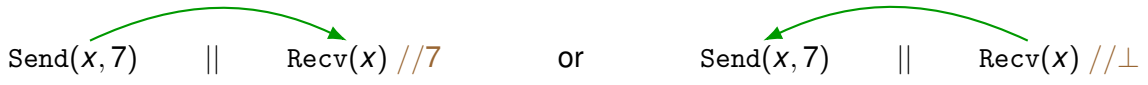
$$\texttt{Send}\colon \mathsf{Loc} \times \mathsf{Val} \to () \quad \text{and} \quad \texttt{Recv}\colon \mathsf{Loc} \to \mathsf{Val} \cup \{\bot\}$$

Specification: Ring Buffer Library

Example 1: Ring Buffer (single-writer-multiple-reader, FIFO)

$$\texttt{Send}\colon \mathsf{Loc} \times \mathsf{Val} \to () \quad \text{and} \quad \texttt{Recv}\colon \mathsf{Loc} \to \mathsf{Val} \cup \{\bot\}$$

$$\texttt{Send}(x, 7) \quad || \quad \texttt{Recv}(x) \; //\text{output?}$$

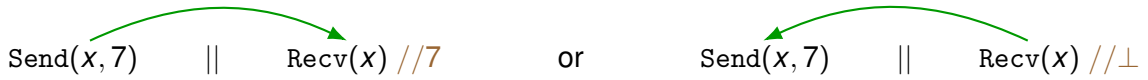


Specification: Ring Buffer Library

Example 1: Ring Buffer (single-writer-multiple-reader, FIFO)

$$\texttt{Send}\colon \mathsf{Loc} \times \mathsf{Val} \to () \quad \text{and} \quad \texttt{Recv}\colon \mathsf{Loc} \to \mathsf{Val} \cup \{\bot\}$$

$$\texttt{Send}(x, 7) \quad || \quad \texttt{Recv}(x) \ //\text{output?}$$



Ring Buffer library spec (wishlist, first draft, version 0)

- Syntax (`Send`, `Recv`)
- Behaviour constraints (e.g. FIFO)
- Synchronisation dependencies ($\longrightarrow$)

Specification: Barrier Library

Example 2: Barrier library

$\texttt{Barr} : \mathsf{Loc} \to ()$

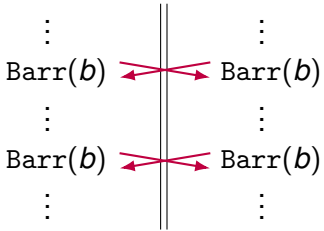
Specification: Barrier Library

Example 2: Barrier library

$\texttt{Barr} : \mathsf{Loc} \to ()$

Thread synchronisation:

All threads reach the barrier before any can continue past.



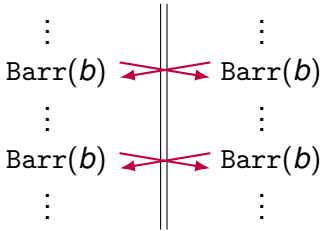
Specification: Barrier Library

Example 2: Barrier library

$\texttt{Barr} : \mathsf{Loc} \to ()$

Thread synchronisation:

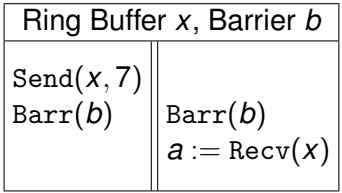
All threads reach the barrier before any can continue past.



Barrier library specification (wishlist, first draft, version 0)

- Syntax (`Barr`)
- Behaviour constraints (e.g. used same number of times by each thread)
- Synchronisation dependencies ($\longrightarrow$)

Execution



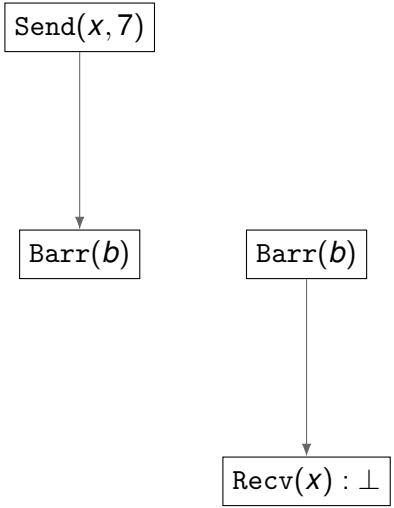
A global execution is allowed if it:

- respects the semantics of every library
- doesn't create dependency cycles

Execution

Ring Buffer x, Barrier b	
$\texttt{Send}(x, 7)$ $\texttt{Barr}(b)$	$\texttt{Barr}(b)$ $a := \texttt{Recv}(x)$

Claim: $a \neq \bot$



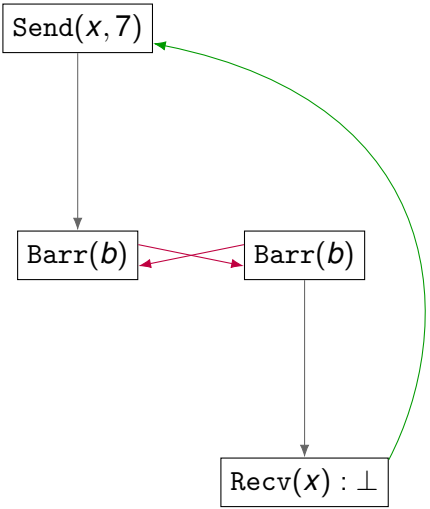
A global execution is allowed if it:

- respects the semantics of every library
- doesn't create dependency cycles

Execution

Ring Buffer x, Barrier b	
$\texttt{Send}(x, 7)$ $\texttt{Barr}(b)$	$\texttt{Barr}(b)$ $a := \texttt{Recv}(x)$

Claim: $a \neq \bot$



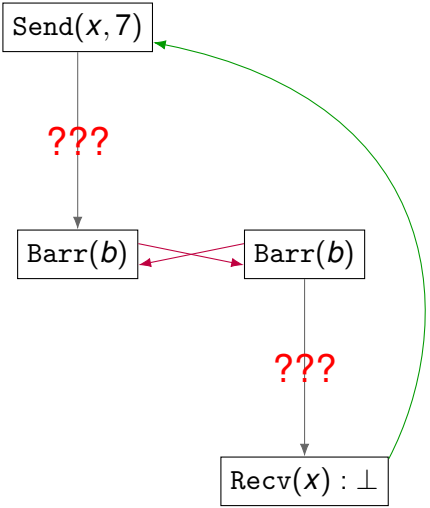
A global execution is allowed if it:

- respects the semantics of every library
- doesn't create dependency cycles

Execution (First Try)

Ring Buffer x, Barrier b	
$\texttt{Send}(x, 7)$ $\texttt{Barr}(b)$	$\texttt{Barr}(b)$ $a := \texttt{Recv}(x)$

Claim: $a \neq \bot$



A global execution is allowed if it:

- respects the semantics of every library
- doesn't create dependency cycles

Program Order Dependencies???

Remember the first example:

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

$z = 1$ ✓

Program order is **not** always respected...

When is it respected?

Stamp ordering for RDMA

Previous documented hardware (re)orderings:

Second Stamp

First Stamp			Ordered?	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$_n$		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

11 **stamps** (**categories** of operations) define when ordering is preserved.

Program Order Dependencies

Remember the first example:

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

$z = 1$ ✓

Program order is **not** always respected...

Reorderings for (independent) libraries?

Program Order Dependencies

Remember the first example:

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

$z = 1$ ✓

Program order is **not** always respected...

Reorderings for (independent) libraries?

Mowgli solution:

- Use hardware behaviours as categories (stamps).
- Libraries specify the effects of each function.

Mowgli Ring Buffer library specification

- like before: syntax, semantics, $\longrightarrow$
- `Send` is in the "NIC write" category
- `Recv` is in the "CPU read" category

Mowgli Barrier library specification

- like before: syntax, semantics, $\longrightarrow$
- `Barr` has Fence and Read behaviours

Mowgli Execution (Fixed)

Ring Buffer x, Barrier b	
$\texttt{Send}(x, 7)$ $\texttt{Barr}(b)$	$\texttt{Barr}(b)$ $a := \texttt{Recv}(x)$

Claim: $a \neq \bot$

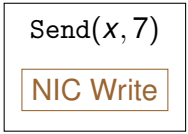
- like before: syntax, semantics, $\longrightarrow$
- `Send` has a NIC write behaviour
- `Recv` has a CPU read behaviour

- like before: syntax, semantics, $\longrightarrow$
- `Barr` has Fence and Read behaviours

Mowgli Execution (Fixed)

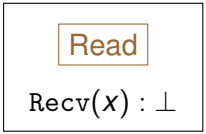
Ring Buffer x, Barrier b	
$\texttt{Send}(x, 7)$ $\texttt{Barr}(b)$	$\texttt{Barr}(b)$ $a := \texttt{Recv}(x)$

Claim: $a \neq \bot$



- like before: syntax, semantics, $\longrightarrow$
- `Send` has a NIC write behaviour
- `Recv` has a CPU read behaviour

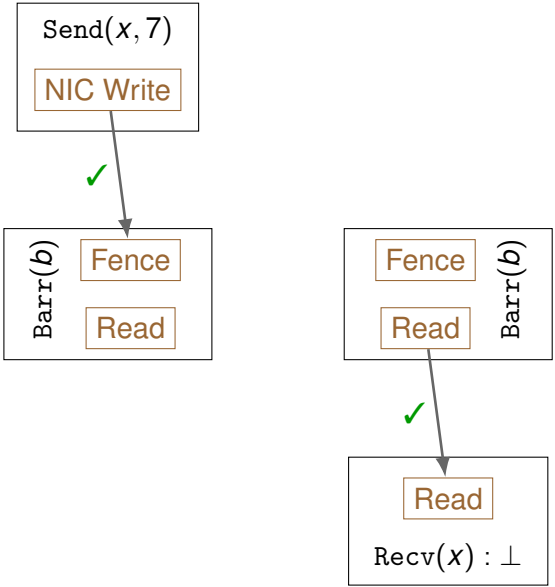
- like before: syntax, semantics, $\longrightarrow$
- `Barr` has Fence and Read behaviours



Mowgli Execution (Fixed)

Ring Buffer x, Barrier b	
$\texttt{Send}(x, 7)$ $\texttt{Barr}(b)$	$\texttt{Barr}(b)$ $a := \texttt{Recv}(x)$

Claim: $a \neq \bot$



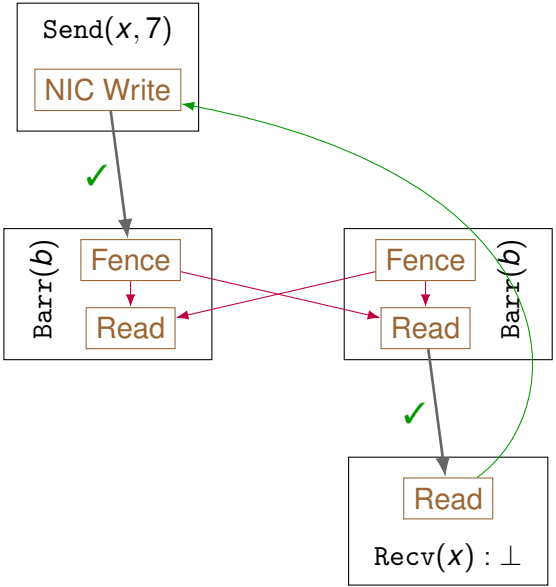
- like before: syntax, semantics, $\longrightarrow$
- `Send` has a NIC write behaviour
- `Recv` has a CPU read behaviour

- like before: syntax, semantics, $\longrightarrow$
- `Barr` has Fence and Read behaviours

Mowgli Execution (Fixed)

Ring Buffer x, Barrier b	
$\texttt{Send}(x, 7)$ $\texttt{Barr}(b)$	$\texttt{Barr}(b)$ $a := \texttt{Recv}(x)$

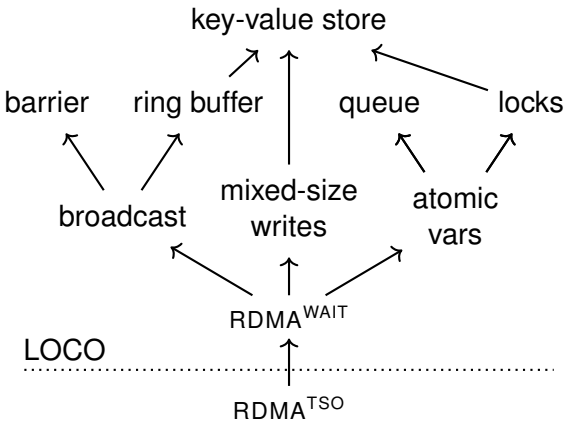
Claim: $a \neq \bot$



- like before: syntax, semantics, $\longrightarrow$
- `Send` has a NIC write behaviour
- `Recv` has a CPU read behaviour

- like before: syntax, semantics, $\longrightarrow$
- `Barr` has Fence and Read behaviours

Verifying Library Implementations



Libraries can be implemented using other libraries

Programs can use a set Λ of libraries

Inlining an implementation I (of library L) **should be fine**

Implementation Soundness

How do we prove correct an implementation I for a library L?

- **Soundness** [general, hard to show]:
  Outcomes of the implementation are valid, in **all** contexts:
  $\forall \Lambda, P.\ \texttt{outcome}_{\Lambda}(I(P)) \subseteq \texttt{outcome}_{\Lambda \uplus \{L\}}(P)$

Implementation Soundness

How do we prove correct an implementation I for a library L?

- **Soundness** [general, hard to show]:
  Outcomes of the implementation are valid, in **all** contexts:
  $\forall \Lambda, P.\ \texttt{outcome}_{\Lambda}(I(P)) \subseteq \texttt{outcome}_{\Lambda \uplus \{L\}}(P)$

- **Local soundness** [library specific, easier to show]:
  Promises of the library specification are kept (formal definition is a bit technical).
  Considers I and L in isolation.

- **Mowgli Locality Theorem**:
  Local soundness implies soundness.

Implementation Soundness

How do we prove correct an implementation I for a library L?

- **Soundness** [general, hard to show]:
  Outcomes of the implementation are valid, in **all** contexts:
  $\forall \Lambda, P.\ \texttt{outcome}_{\Lambda}(I(P)) \subseteq \texttt{outcome}_{\Lambda \uplus \{L\}}(P)$

- **Local soundness** [library specific, easier to show]:
  Promises of the library specification are kept (formal definition is a bit technical).
  Considers I and L in isolation.

- **Mowgli Locality Theorem**:
  Local soundness implies soundness.

Programmers prove local soundness, Mowgli gives soundness for free.

Conclusion

Summary:

- Generalised RDMA^{TSO} to $\text{RDMA}^{\text{WAIT}}$ as a modular alternative
- LOCO: High-performance Modular RDMA Libraries
- Mowgli: Framework for Libraries in (very) weak settings such as RDMA
- Local soundness result for simpler verification
- Specified and verified a large fragment of LOCO using Mowgli

Next: extension to RDMA Atomic operations (ESOP 2026)

Conclusion

Summary:

- Generalised RDMA^{TSO} to $\text{RDMA}^{\text{WAIT}}$ as a modular alternative
- LOCO: High-performance Modular RDMA Libraries
- Mowgli: Framework for Libraries in (very) weak settings such as RDMA
- Local soundness result for simpler verification
- Specified and verified a large fragment of LOCO using Mowgli

Next: extension to RDMA Atomic operations (ESOP 2026)

Thanks for listening!

Poll vs Wait Semantics

Base RDMA (RDMA^{TSO}):

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

$z = 1$ ✓

$x = 0$	$z = 0$
$\widetilde{z} := x$ `Poll()` $x := 1$	

$z = 1$ ✗

$x = 0$	$z = 0$
$\vdots$ $\widetilde{z} := x$ `Poll()` $x := 1$	

$z = 1$ ✓

Poll vs Wait Semantics

Base RDMA (RDMA^{TSO}):

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

$z = 1$ ✓

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

$z = 1$ ✗

$x = 0$	$z = 0$
$\vdots$ $\widetilde{z} := x$ $\texttt{Poll}()$ $x := 1$	

$z = 1$ ✓

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

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

$z = 1$ ✓

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

$z = 1$ ✗

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

$z = 1$ ✗