Modeling and Analysis of Remote Memory Access Programming

Andrei Marian Dan
andrei.dan@inf.ethz.ch
ETH Zurich, Switzerland

Patrick Lam
patrick.lam@uwaterloo.ca
University of Waterloo, Canada

Torsten Hoefler
torsten.hoefler@inf.ethz.ch
ETH Zurich, Switzerland

Martin Vechev
martin.vechev@inf.ethz.ch
ETH Zurich, Switzerland

Abstract
Recent advances in networking hardware have led to a new generation of Remote Memory Access (RMA) networks in which processors from different machines can communicate directly, bypassing the operating system and allowing higher performance. Researchers and practitioners have proposed libraries and programming models for RMA to enable the development of applications running on these networks.

However, the memory models implied by these RMA libraries and languages are often loosely specified, poorly understood, and differ depending on the underlying network architecture and other factors. Hence, it is difficult to precisely reason about the semantics of RMA programs or how changes in the network architecture affect them.

We address this problem with the following contributions: (i) a coreRMA language which serves as a common foundation, formalizing the essential characteristics of RMA programming; (ii) complete axiomatic semantics for that language; (iii) integration of our semantics with an existing constraint solver, enabling us to exhaustively generate coreRMA programs (litmus tests) up to a specified bound and check whether the tests satisfy their specification; and (iv) extensive validation of our semantics on real-world RMA systems. We generated and ran 7,441 litmus tests using each of the low-level RMA network APIs: DMAPP, VPI Verbs, and Portals 4. Our results confirmed that our model successfully captures behaviors exhibited by these networks. Moreover, we found RMA programs that behave inconsistently with existing documentation, confirmed by network experts.

Our work provides an important step towards understanding existing RMA networks, thus influencing the design of future RMA interfaces and hardware.

Keywords Memory Model

Categories and Subject Descriptors B.3.3 [Performance Analysis and Design Aids]: Formal models

1. Introduction
Large-scale parallel systems are gaining importance for data center, big data, and scientific computations. The traditional programming models for such systems are message passing (e.g., through the Message Passing Interface—MPI) and TCP/IP sockets (as used by Hadoop, MapReduce, or Spark).

These models were designed for message-based interconnection networks such as Ethernet. Remote Direct Memory Access (RDMA) network interfaces, which have been used in High-Performance Computing for years, offer higher performance at a comparable cost to Ethernet and are finding quick adoption in modern datacenters. To extract the highest performance from such modern interconnects, programmers need to use Remote Memory Access (RMA) programming interfaces, which are replacing traditional message passing models.

Key Benefits of RMA. RMA enables direct access to remote memory through the network interface. RMA bypasses the operating system and the CPU, enabling low latencies and high bandwidth—remote access times of less than 1µs are possible today (Gerstenberger et al. 2013). Since the hardware implementation in the network card is a simple set of queues, RMA technology is widely supported; it is available for InfiniBand (The InfiniBand Trade Association 2004), Blue Gene/P (Allen et al. 2001), Blue Gene/Q (Chen et al. 2011), IBM PERCS (Arimilli et al. 2010), and Cray’s Gemini and Aries networks (Alverson et al. 2010; Faanes et al. 2012). RMA-capable hardware is now in the same price range as standard Ethernet network cards while providing higher performance.
**RMA Programming.** At the lowest level, RMA networks are programmed through user-level libraries that directly communicate with the hardware. These libraries provide calls to read and write remote memory locations as well as various forms of synchronization that a program can use. Therefore, programming RMA systems is conceptually similar to shared memory systems. The main differences are that 1) RMA systems do not offer atomicity by default (Dunning et al. 1998) and 2) the global address space is partitioned such that each network endpoint owns a fixed address range. Several programming systems embrace remote memory access (RMA) functionality (Numrich and Reid 1998; UPC Consortium 2005; Hoefler et al. 2013; Valiev et al. 2010).

**RMA-Based Libraries.** RMA library interfaces are specific to network technologies and include InfiniBand’s Open Fabrics Enterprise Distribution (OFED (OpenFabrics Alliance (OFA) 2014)), Cray’s uGNI and DMAPP (Cray Inc. 2014), the Portals 4 network programming interface (Barrett et al. 2012) and IBM’s Parallel Active Messaging Interface (PAMI (Kumar et al. 2012)). Many middleware applications, such as Hadoop (Islam et al. 2012), call these interfaces directly. Unfortunately, most of these interfaces only specify loose memory semantics. No standard interface or memory model has been established yet, e.g., RMA library interfaces do not guarantee that all accesses are atomic, and some implementations lead to undefined results for overlapping accesses, while others require explicit operations to guarantee visibility.

**Our Work** To address these challenges, in this work, we define the first formal model, coreRMA, which cleanly captures essential characteristics of RMA programming. coreRMA serves as a basis for specifying the constructs of future RMA languages and libraries. We encoded our semantics using a state-of-the-art relational solver, enabling programmers and network experts to quickly experiment with RMA configurations and scenarios. Finally, based on our semantics, we exhaustively (up to a bound) generated test cases which conformed to arcane low-level real-world APIs of RMA networks, executed them, and found inconsistencies.

**Main Contributions** Our main contributions are:

- The first formal axiomatic definition of RMA semantics, coreRMA, a common foundation formalizing essential characteristics of RMA networks. These characteristics include network routing and asynchronous execution.
- An implementation of the coreRMA model in an analysis tool based on relational logic and a validation framework including test generation for real world networks.
- A systematic experimental validation of our model on Cray Aries and InfiniBand, using the DMAPP, Portals 4 and IBV Verbs libraries, which discovered behaviors that contradict both current RMA network documentation and our model, as well as predicted behaviors that never occurred. These inconsistencies were confirmed by RMA network experts.

2. **Overview**

In this section, we provide an intuitive explanation of RMA semantics and illustrate allowed RMA behaviors using examples. Sections 3–5 provide the full coreRMA semantics.

Consider the following RMA program with two processes $P1$ and $P2$. The program has shared variables $x$ belonging to $P1$ and $y$ to $P2$, with initial values 0 and 1, respectively, along with local registers $a$ and $b$. We assume that programs synchronize after setting initial values for their shared variables.

$$
\begin{align*}
X &= 0 & Y &= 1 \\
P1: & \quad P2: \\
X &= \text{get}(Y) \quad Y &= \text{get}(X) \\
a &= X & b &= Y
\end{align*}
$$

This program demonstrates RMA’s remote reads and writes. The first process reads remotely the value of $y$ and stores it in $x$, while the second process reads remotely the value of $x$ and stores it in $y$. Remote accesses are enqueued by a CPU onto its network interface card (NIC), which then executes required remote communication and memory accesses without further CPU involvement. After initiating the remote accesses, each process reads locally the variables $x$ and $y$, respectively, and stores results in registers $a$ and $b$.

To understand this program under sequential consistency, it suffices to consider the interleavings of the actions making up the get statements in each process. Possible outcomes include $a = 0$, $b = 0$; $a = 1$, $b = 1$; and, with non-atomic get statements, $a = 1$, $b = 0$. However, RMA admits additional behaviors, because the local reads are not guaranteed to run after the get statements. Thus, a possible outcome under a non-sequentially-consistent memory model is $a = 0$, $b = 1$. Our axiomatic semantics of RMA enable the prediction of such admissible hardware behaviors and the detection of inconsistencies between the model and the hardware. By generating tests from our model, we have confirmed 135 instances where Cray hardware exhibits behavior that violates its documentation and 13 instances for Portals 4; Section 7 presents those cases. We have also observed that our model is reasonably tight: actual hardware exhibits 90% of the expected outputs from our model.

2.1 **RMA Hardware Model**

Modern commodity and special-purpose high-performance network interfaces can be modeled with an abstract RMA interface. However, detailed memory ordering semantics vary widely between the different network cards, and it is important to understand them to write correct programs.

Figure 2 shows the basic architecture of an RMA system. Operations are issued by a program running on a CPU. When the CPU performs a remote write operation, it instructs the network interface card (NIC) to copy data from local memory at the source to remote memory at the tar-
atomic instructions. The model in this paper applies to any set of atomicity guarantees.

**Ordering.** Ordering of accesses between the same two endpoints is generally not guaranteed. Some networks, such as InfiniBand, maintain the order of either remote reads or writes between the same pair of endpoints. Others, such as Cray’s Gemini or Aries or IBM’s PERCS network, relax the ordering to enable network optimizations such as adaptive routing. Most modern low-diameter topologies require adaptive routing to provide a high global bandwidth (Jiang et al. 2009).

### 2.2 SC Behaviors versus RMA Behaviors

To illustrate the challenges of reasoning about RMA programs, we present several examples which illustrate the intricacies that arise when dealing with RMA behaviors. We show RMA behaviors that differ from sequentially consistent (SC) executions as well as behaviors not exhibited by other memory models studied in the literature such as TSO (Owens et al. 2009), PSO, and RMO (SPARC International 1992) (such models obey local data dependencies; a write to variable \( x \) is visible to subsequent reads from \( x \); this is not the case for RMA, see Section 2.3).

Figure 1 shows possible behaviors of the simple program from the start of the section. We split each get statement into read and write actions. The leftmost behavior from Figure 1 shows \( X = \text{get(Y)} \) from \( P1 \) split into two statements: read \( Y,0 \) where the get statement reads value 0 from \( Y \); and write \( X,0 \) where the get writes 0 to \( X \).

The relation \( \xrightarrow{po} \) represents the program order between the actions. The \( \xrightarrow{hb} \) relation is the happens-before relation, also known as the consistency order: if two actions are ordered by happens-before, then the effects of the first action are visible to the second action. In sequential consistency, program order \( \xrightarrow{po} \) implies happens-before \( \xrightarrow{hb} \). Hence, under SC, if an action appears in the program before another action, the effects of the first action are guaranteed to be visi-
ble by the second action. The \( \rightarrow \) relation indicates the write action from which a read action reads from. Figure 1 shows that read and write actions constituting a `get` statement are always ordered by both `po` and `hb`. However, the local read action is ordered after the `get` action only by relation `po`. This reflects the fact that, under RMA, the effects of a `get` are not guaranteed to be visible to subsequent local actions.

### Sequentially Consistent Behaviors

Reasoning about concurrent programs requires considering (or ruling out) all possible interleavings; we continue by enumerating interleavings. One case, furthest to the left in Figure 1, is when `Y = get(X)` runs before `X = get(Y)`. At the end of this execution, `a` and `b` are both 0, because the local read statements from `x` and `y` read the most recent writes.

In the second SC case (also second from left in Figure 1), `X = get(Y)` runs before `Y = get(X)`. Now, both `a` and `b` get 1. Assuming sequential consistency and atomicity of the `get` statement, there are only two possible outcomes of the program: the pair of variables `(a, b)` can have either the values `(0, 0)` or `(1, 1)`.

A third possibility we allow as sequentially consistent behavior is when the constituent sets of actions of the `get` statements are not executed atomically. For example, the `read Y` and `write X` actions from `X = get(Y)` may be interleaved with the `read X` and `write Y` actions from `Y = get(X)`. It is thus possible that both `get` read actions correspond to initial values. This leads to `(a, b)` having values `(1, 0)`, shown as the third SC behavior in Figure 1.

### Non-Sequentially Consistent Behavior

When we execute the program on an RMA network, we observe additional non-sequentially consistent behaviors. An example of such a behavior (shown rightmost in Figure 1) leaves `(a, b)` with the values `(0, 1)`. Since the local reads in each process are not ordered by `hb` after the writes of the `get` statements, these local reads may read from the initial values of the variables. This execution leads to the values `(0, 1)` for `(a, b)` and is a valid execution under RMA.

### 2.3 Out of Order Execution

To provide additional intuition for the RMA semantics, we continue with more examples permitting RMA behaviors not possible under sequential consistency or other hardware memory models (e.g., x86 TSO, PSO, RMO). Figure 3 summarizes these examples; we show, for each example, the source code and one possible behavior. Statements are on the left and the corresponding actions are on the right. See Table 3 for the translation from statements to actions.

#### a) `get`: out of order execution.

In example a) of Figure 3, process 1 hosts shared variable `x` and the second process hosts shared variable `y`. Both variables are initialized to 0. In the sequentially consistent (SC) case, we treat `put` and `get` simply as a shared write and a shared read respectively.

The second process does not execute any statements. Under sequential consistency, when the program terminates, local register `a` is 1. However, under RMA, `a` can be 0, 1, or undefined (denoted as `⊤`). Variable `a` may be 0 because the statement `x = get(y)` may complete after `x = 0`. Variable `a` may be `⊤` because the write `x = get(y)` can happen concurrently with `x = 1` and the atomicity of these accesses is not guaranteed. In the diagram, the `hb` relation indicates that the effects of action `read y, 0` are visible before action `write x, 0` is executed and, similarly, the effects of `write x, 1` are visible before the action `read x, 0` is executed. Since `write x, 0` is not ordered by `hb` with `write x, 1`, those two actions may be executed in any order.

### Comparing RMA to TSO, PSO, and RMO

This first example also illustrates a case where programs under RMA allow behaviors that are not possible in other weak memory consistency models, such as RMO, PSO, or x86 TSO. Consider RMO, the most relaxed (permissive) memory model of these three. In RMO, writes to the same variable issued by a process are always ordered. For example, `x = 1` is ordered after `x = get(y)`, so that the read `a = x` can return only 1. However, recall that under RMA, register `a` can be `0`, which is not possible under RMO, PSO, or x86 TSO.

#### b) `put`: out of order execution.

In example b) of Figure 3, once again, the first process hosts the shared variable `x` and the second process hosts `y`, and both are initialized to 0. Statement `put(y, x)` in the first process means that `y` gets the value of `x`. Under SC, upon termination, local variable `b` is always 0. Under RMA, `b` can also be 1, because `put(y, x)` may complete after the write `x = 1`. As in the first example, the statements of the first process can execute simultaneously, so the final value of `b` can also be `⊤`. Again, output `b = 1` is not allowed under other relaxed buffered memory models, such as RMO, PSO, or x86 TSO. In this case, `read x, 0` could not read from an action which occurs after itself under `hb` (namely `write x, 1`).

#### c) `put-get sequence`.

In example c) of Figure 3, variable `x` is initialized to 1 and `y` to 0. The `flush` statement ensures that the `get` and `put` statements complete before executing `c = x`. This example shows the effect of `ordering` the accesses between the same two endpoints. Under SC, upon termination, local register `c` is 1. Under RMA, if the accesses of the first process to the memory of the second process are not ordered, the final value of `c` may also be 0: the statement `x = get(y)` is executed before `put(y, x)`. However, if the network ensures ordered accesses between the same two endpoints (referred to as `in-order routing`), discussed later, then the sequence `put-get` is ordered and the value 0 is not possible for `c` (the `get` statement will read the value written by the previous `put` statement). Finally, `c` may be undefined due to a race between non-atomic `read`s and `write`s on `x`.

#### d) `get-put sequence`.

In example d) of Figure 3, `x` is initially 1 and `y` is 0. First, `P1` reads the value of `y` and stores
Figure 3: Example programs show behaviors allowed by coreRMA that are not allowed under sequential consistency. Table 3 provides our translation from get and put statements into read and write actions.

it in \( x \). Next, the value of \( x \) is written to \( y \). After the flush, the value of \( y \) is read again and stored to \( x \). The interesting behavior in this example is that, even if the ordering of accesses between the same two endpoints is guaranteed by the network, the value of \( d \) can be 1. This result is counter-intuitive because it appears that the get and put statements before the flush are executed in reverse order, even if the two statements are accesses between the same two endpoints and the network guarantees order between such accesses. This type of behavior is allowed by the RMA networks, and our model enables users to reason about this behavior.

3. The coreRMA Language

We start our formal description of RMA semantics by presenting the statements of our coreRMA language (see Table 1). These statements include the core RMA primitives and are sufficiently expressive to capture the essence of RMA programs. Our description of RMA behaviors builds on the semantics of these statements.

3.1 RMA-Based Programming Models

Remote Memory Access (RMA) languages provide interfaces to emerging RMA networks. These languages are gaining popularity in HPC and finding adoption in datacenter en-

environments (Dragojević et al. 2014; Poke and Hoefler 2015). Successful complex applications such as NWChem (Valiev et al. 2010) rely solely on RMA programming.

A number of RMA languages take advantage of RMA hardware acceleration. A key difficulty of implementing RMA languages lies in using the underlying RMA library as efficiently as possible, yet legally. Underlying RMA programming models are still in active development and far from understood: MPI One Sided, for example, was re-
vamped completely in MPI-3.0 (2012) and continues to evolve towards MPI-4.0.

For coreRMA, we identified 5 primitive remote-access statements: put, get, rga, cas, and flush. These statements implement the constructs found in higher-level languages and libraries. Table 2 shows mappings from constructs in Cray’s DMAPP API, OFED’s IB API, Portals 4, UPC, Fortran 2008, and MPI-3 RMA, to coreRMA primitives.

3.2 coreRMA

We next explain the components of our coreRMA language.

Processes, Registers, and Memory Locations. A coreRMA program consists of a finite set of processes $\mathit{Processes} = \{p_1, p_2, \ldots, p_N\}$. coreRMA supports a single process per computation node. Each $p \in \mathit{Processes}$ has a set $\mathit{Registers}[p]$ of local registers. Local registers cannot be accessed from other processes. The set $\mathit{Memory}[p]$ denotes the memory locations of process $p$, which are accessible to all processes. The set of all remotely accessible memory locations is $\mathit{Memory} = \bigcup_{p \in \mathit{Processes}} \mathit{Memory}[p]$. We use the terms memory location and variable interchangeably.

Local Statements. A local statement can only read or write variables that belong to the process that executes the statement. Using a local statement, process $p$ can access variables in $\mathit{Memory}[p]$. It is local in the sense that it does not access the memory of other processes. An access to a variable can be either atomic or non-atomic. We use the symbol $\ast$ to range over both types of accesses, that is, $\ast \in \{a, n\}$, where $a$ stands for atomic and $n$ stands for non-atomic.

Let $X$ be a variable of $\mathit{Memory}[p]$, $u$ be a register of $\mathit{Registers}[p]$, and $\exp \mathit{e}$ an expression containing registers and numerical values. The two kinds of local statements are local read ($u \leftarrow X$) and local write ($X \leftarrow \exp$).

Remote Statements. A remote statement can read or write any memory location. Using a remote statement, process $p$ can access any variable in $\mathit{Memory}$. The notation $\ast$ also indicates atomicity here. Since remote put and remote get operations have two memory interactions, subscript $\ast$ indicates atomicity for each interaction. Our language and formal semantics support all 4 atomicity combinations. Specific networks support a subset of these combinations.

Remote statements are performed asynchronously. When a process executes a remote statement, it instructs the network interface card to perform the necessary read and write operations, and continues immediately.

Let $Z_{\mathit{dst}}$ be a variable from $\mathit{Memory}$ ($\mathit{dst} \in \mathit{Processes}$ is the target process). Let $X, Y$ be variables from $\mathit{Memory}[p]$.

Remote get: $X \leftarrow \mathit{get}(Z_{\mathit{dst}})$. Process $p$ reads $Z_{\mathit{dst}}$ and writes it to its local memory location $X$.

Remote put: $\mathit{put}(Z_{\mathit{dst}}, X)$. Process $p$ reads local memory location $X$ and writes it to $Z_{\mathit{dst}}$ at process $\mathit{dst}$.

Remote get accumulate: $X \leftarrow \mathit{rga}(Z_{\mathit{dst}}, Y)$. Without loss of generality, consider accumulate function $+$ (addition). Process $p$ reads the value of local memory location $Y$. Next, it uses a read-write operation to read $Z_{\mathit{dst}}$ and write back the sum of $Y$ and $Z_{\mathit{dst}}$. Finally, it writes the value of $Z_{\mathit{dst}}$ that was read initially to local memory location $X$.

Compare and swap: $X \leftarrow \mathit{cas}(Z_{\mathit{dst}}, X, W)$. Process $p$ reads the values of local memory locations $Y$ and $W$. Next, it uses a read-write operation on $Z_{\mathit{dst}}$ to read its value and, if $Z_{\mathit{dst}}$ and $Y$ are equal, it writes the value of $W$ to $Z_{\mathit{dst}}$ else it leaves the value of $Z_{\mathit{dst}}$ unchanged. Finally, it writes the value of $Z_{\mathit{dst}}$ that was read initially to $X$.

Flush Statement. The flush statement $\mathit{flush}(\mathit{dst})$ waits until all remote operations from the process executing the flush to the process $\mathit{dst}$ complete.

4. From Statements to Actions

RMA statements, described in Section 3, comprise one or more actions (e.g., a remote put performs both a read and a write). We now define how we decompose coreRMA statements into actions. The translation from statements to actions enables uniform reasoning about programs by allowing us to describe (in Section 5) the axiomatic semantics of the language on the set of actions.

Types of Actions. An action has one of six types: local write ($w$), local read ($r$), external read (ew), external write (ew), external read-write (ew), and flush (flush). Actions which write to or read from memory carry a star, indicating their atomicity. We define 3 disjoint sets of action types:

Local actions: Local contains local actions: \{r, w\}.

External actions: External contains remote actions that interact with memory: \{ew, ew, ew\}.

Flush actions: Flush contains flush statements, which do not interact with memory: \{flush\}.

Local and external actions perform operations on memory while flush actions constrain ordering. Actions may be readers or writers. Set Reader = \{r, ew, ew\} contains atomic and non-atomic local read and external read and read-actions, while set Writer = \{w, ew, ew\} contains atomic and non-atomic local write and external write and external read-write actions. ew actions perform both reads and writes and hence belong to both sets.

Attributes of Actions. We define auxiliary functions:

- $\mathit{src}$: origin/source process, which originates the action,
- $\mathit{dst}$: destination process, which executes the action,
- $\mathit{loc}$: memory location accessed by a reader action,
- $\mathit{w}$: memory location modified by a write action.

In the context of a particular statement, we denote the executing process by $p$. 
Table 2: coreRMA primitives and corresponding constructs in popular RMA languages.

<table>
<thead>
<tr>
<th>RMA</th>
<th>put</th>
<th>get</th>
<th>flush</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMAPP</td>
<td>dmapp_put_nbi</td>
<td>dmapp_get_nbi</td>
<td>dmapp_gsync_wait</td>
</tr>
<tr>
<td>OFED (IB)</td>
<td>ibv_wr_rdma_write</td>
<td>ibv_wr_rdma_read</td>
<td>ibv_reg_notify_cq</td>
</tr>
<tr>
<td>Portals 4</td>
<td>PtIPut</td>
<td>PtIGet</td>
<td>PtICWait</td>
</tr>
<tr>
<td>UPC</td>
<td>upc_memput</td>
<td>upc_memget</td>
<td>upc_fence</td>
</tr>
<tr>
<td>Fortran 2008</td>
<td>assignment</td>
<td>assignment</td>
<td>sync_memory</td>
</tr>
<tr>
<td>MPI-3 RMA</td>
<td>MPI_Put</td>
<td>MPI_Get</td>
<td>MPI.Win_flush</td>
</tr>
</tbody>
</table>

Table 3: Translation scheme from statements into sets of actions, and corresponding attributes.

| X_n = expr | \{l\} |
| l: type = w, src = p, dst = p, w_{loc} = X |
| u = X_a | \{l\} |
| l: type = r_{\text{t}}, src = p, dst = p, r_{loc} = X |
| X_a = get (z_{\text{dst}}) | \{e_1, e_2\} |
| e_1: type = r_{\text{t}}, src = p, dst = dst, r_{loc} = Z |
| e_2: type = e_{\text{w}}, src = p, dst = p, w_{loc} = X |
| put (z_{\text{dst}}, X_a) | \{e_1, e_2\} |
| e_1: type = e_{\text{r}}, src = p, dst = p, r_{loc} = X |
| e_2: type = e_{\text{w}}, src = p, dst = dst, w_{loc} = Z |
| X_n = rga (z_{\text{dst}}, Y_n) | \{e_1, e_2, e_3\} |
| e_1: type = e_{\text{r}}, src = p, dst = p, r_{loc} = Y |
| e_2: type = e_{\text{w}}, src = p, dst = dst, w_{loc} = Z |
| e_3: type = e_{\text{w}}, src = p, dst = p, w_{loc} = X |
| flush (dst) | \{f\} |
| f: type = flush, src = p, dst = dst |

4.1 Translation of Statements to Actions

Let \[\cdot\]: Statement \rightarrow P(\text{Action})\ map statements to generated actions. Table 3 illustrates this function for coreRMA statements. Without loss of generality, we illustrate only one choice of atomicity properties per statement. Table 4 shows paradigmatic statements and their translations into sets of actions, along with relevant ordering relations.

For a local non-atomic write statement \(X_n = \text{expr}\), the corresponding action \(f\) has type \(w\) (non-atomic local write action), the origin and destination of the action are both \(p\), and the write location is \(X\). Atomic local write statements (not shown) only differ in action type, which would be \(w_{\text{loc}}\).

Moving on to remote statements, the remote get statement \(X_n = \text{get (z_{\text{dst}})}\) produces two actions: \(e_1\) and \(e_2\). The first column of Table 4 shows one translation. This translation also includes the ordering relations \(\overset{\text{po}}{\rightarrow}\) and \(\overset{\text{hb}}{\rightarrow}\), which are formally introduced in Section 5. External action \(e_1\) has type non-atomic external read \((e_{\text{r}})\). The destination process of \(e_1\) is \(dst\), the process which owns \(z\). The read location is \(Z\). External action \(e_2\) has type atomic external read \((e_{\text{w}})\). Its destination process is \(p\) (the process executing the statement) and the write location is \(X\). Remote puts are analogous.

Remote get accumulate \(X_n = \text{rga (z_{\text{dst}}, Y_n)}\) generates three external actions: a read \(e_1\) from variable \(Y\), a read-write action \(e_2\) which reads the value of \(z_{\text{dst}}\) and writes back the sum of the two reads, and a write to \(X\), \(e_3\), which writes the same value as the read-write action. The second column of Table 4 shows the translation of an rga statement.

A compare and swap statement generates four external actions: two external reads, an external read-write and an external write action (third column of Table 4).

At the bottom of Table 3 we show a flush action corresponding to statement flush (dst). The origin of action \(f\) is the process executing the flush statement (denoted \(p\)) and the destination is the target of the flush, \(dst\).

Atomicity Properties. Decomposing statements into actions enables fine-grained specification of atomicity properties. Atomicity properties of statements can either be specified by a language or ensured by the RMA network specification. Our model handles all possible atomicity properties.

5. Axiomatic Semantics of coreRMA

We next present the formal axiomatic semantics of the coreRMA language. We designed these semantics to capture common behaviors in RMA networks yet to be flexible enough to allow for expressing specifics of real world net-
The rule for remote put and remote get.

Rule for remote put and remote get. We introduce predicates $rp$ and $rg$ to identify the two component actions of

Table 5: Relations and functions that define an execution.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$po$</td>
<td>Program order: relates all pairs of actions of the same process; does not relate actions from different processes. For each process $p$, $po$ is a total order for all actions in $p$. Acyclic, transitively closed, and not reflexive.</td>
</tr>
<tr>
<td>$hb$</td>
<td>Happens before: when $a_1 \stackrel{hb}{\rightarrow} a_2$, the effects of $a_1$ are guaranteed visible to $a_2$. Acyclic, transitively closed, and not reflexive.</td>
</tr>
<tr>
<td>$rf$</td>
<td>Reads from: associates each $w \in Writer$ with the variable; at least one of the two actions is non-atomic. If $r \in Reader$ operations that read the value written by $w$: $w \stackrel{rf}{\rightarrow} r$. Actions $w$ and $r$ must target the same variable.</td>
</tr>
<tr>
<td>$rval$</td>
<td>Read value: for all $r \in Reader$, $rval(r)$ returns the value read by $r$.</td>
</tr>
<tr>
<td>$wval$</td>
<td>Write value: for all $w \in Writer$, $wval(w)$ returns the value written by $w$.</td>
</tr>
</tbody>
</table>

works (as we present in Section 7). In our semantics, we represent a program execution with a tuple of the form:

$$\langle Action, \stackrel{po}{\rightarrow}, \stackrel{hb}{\rightarrow}, \stackrel{rf}{\rightarrow}, rval, wval \rangle.$$ 

Table 5 presents the meanings of these relations and functions. Each of the three relations $\stackrel{po}{\rightarrow}$, $\stackrel{hb}{\rightarrow}$, and $\stackrel{rf}{\rightarrow}$ is acyclic in a valid RMA program execution. Variables’ initial value assignments are modeled as write actions to these variables, ordered by $\stackrel{hb}{\rightarrow}$ before all program actions.

5.1 Relations over Actions

The $\stackrel{po}{\rightarrow}$ relation is determined from a program’s source code. The relation $\stackrel{hb}{\rightarrow}$ is derived from this section’s inference rules. In general, many relations $\stackrel{rf}{\rightarrow}$ are possible; each such relation encapsulates the data-flow choices taken in an execution. Valid reads-from relations must not induce cycles in $\stackrel{hb}{\rightarrow}$. The two remaining functions, $rval$ and $wval$, depend on the choice of $\stackrel{rf}{\rightarrow}$ and the axiomatic rules, particularly the conflict semantics described in Section 5.3.

5.2 Conflicts

Two actions happen in parallel if they are not ordered by happens-before:

$$a \parallel b \equiv \neg(a \stackrel{hb}{\rightarrow} b) \land \neg(b \stackrel{hb}{\rightarrow} a).$$

A conflict between actions $a$ and $b$, denoted $conflict(a, b)$, occurs when $a \parallel b$; $a$ and $b$ are directed towards the same variable; at least one of the two actions is in $Writer$; and at least one of the two actions is non-atomic. If $r$ is a read action, $conflict(r)$ is true iff there exists a write action $w$ such that $conflict(r, w)$. Similarly, $wconflict(w)$ is true iff there exists a write action $w'$ such that $conflict(w, w')$.

5.3 Rules

We next explain the axiomatic rules in Figure 4. Our descriptions use $Local$, $External$, and $Flush$ from Section 4. Our rules fall into two categories: most rules establish relation $\stackrel{hb}{\rightarrow}$, and the remaining rules define the read and written values $rval$ and $wval$. Subject to some exceptions, documented in Section 7, these rules are consistent with the behavior and documentation of all of the RMA networks that we studied.

Rules for the Reads-from Relation. The reads-from relation influences the happens-before relation through rules $R1$ and $R2$. Let sets $AWriter$ and $AReader$ denote the sets of atomic writes and reads respectively. The first rule ($R1$ in Figure 4) states that if there exist two atomic writes $w_1$ and $w_2$ to the same variable, ordered by $\stackrel{hb}{\rightarrow}$, and if there exists an atomic read action $r$ that reads from $w_1$, then $r$ is also ordered before $w_2$ by $\stackrel{hb}{\rightarrow}$. This rule is not specific to RMA; it also holds for sequential consistency.

The second rule involving the reads-from relation ($R2$) states that an atomic read $r$ is ordered by $\stackrel{hb}{\rightarrow}$ after an atomic write $w$ if $r$ reads from $w$. This ensures that the subset of the reads-from relation between atomic reads and writes is included in the happens-before relation.

Conflict Semantics. Rules $no-C$ and $C$ state permissible behaviors in the absence and presence of conflicts under RMA. If a reader action $r$ is conflict free and the writer action $w$ from which it reads $(w \stackrel{rf}{\rightarrow} r)$ is write-conflict free, then the value read by $r$ is equal to the value written by $w$ ($no-C$). Otherwise, the value read by $r$ is undefined ($C$).

Rule for In-Order Routing Guarantees. Let $Remote$ be the subset of the external actions $External$ containing only the external actions that interact with variables stored at the target process. That is, $Remote$ contains: for remote put statements, the external write action $ew$; for remote get, the external read $er$; and, for remote get accumulate and compare and swap, the external read-write $erw$.

Our formal model captures in-order routing via rule $IR$: remote actions ordered by $\stackrel{po}{\rightarrow}$ are also ordered by $\stackrel{hb}{\rightarrow}$.

Rules Corresponding to the Flush Statement. The rules $F1$–$F3$ describe the relations between flush actions and others. Rule $F1$ states that if a flush action $f$ is ordered with program order before a local action $l$, then $f$ also is ordered by happens-before before $l$.

Given $Remote$ action $e$, let $eactions(e)$ be the set of external actions generated by $e$’s containing statement. For example, if $e$ is a read action generated by a $get$ statement, then $eactions(e)$ contains $e$ plus the companion write action generated by that $get$ statement.

Rule $F2$ states that if a $Remote$ action $e$ is ordered by $\stackrel{po}{\rightarrow}$ before a flush $f$, and if $e$ and $f$ target the same process, then all actions $eactions(e)$ are ordered by $\stackrel{hb}{\rightarrow}$ before $f$. Rule $F3$ is symmetric to $F2$, but imposes $\stackrel{hb}{\rightarrow}$ on successors rather than predecessors.

Rule for Remote Put and Remote Get. We introduce predicates $rp$ and $rg$ to identify the two component actions of
Reads-from relation:  
\[ r \in AReader \quad w_1, w_2 \in AWriter \quad w_1 \xrightarrow{rf} r \quad w_1 \xrightarrow{hb} w_2 \quad w_{\text{cov}}(w_1) = w_{\text{cov}}(w_2) \]  

Conflicts:  
\[ r \in Reader \quad w \in Writer \quad w \xrightarrow{rf} r \quad \neg \text{conflict}(r) \quad \neg \text{wconflict}(w) \]  
\[ r = \top \quad \text{Conflict}(r) \quad \wedge \text{wconflict}(w) \]  

In-order routing:  
\[ e_1, e_2 \in \text{Remote} \quad \text{dst}(e_1) = \text{dst}(e_2) \quad e_1 \xrightarrow{po} e_2 \]  

Remote Get Accumulate:  
\[ er, erw, ew \in \text{External} \quad \text{rga}(er, erw, ew) \]  
\[ er \xrightarrow{po, hb} erw \xrightarrow{po, hb} ew \wedge w_{\text{cov}}(erw) = w_{\text{cov}}(er) \wedge w_{\text{cov}}(ew) = w_{\text{cov}}(erw) \]  

Remote Compare And Swap:  
\[ er_1, er_2, erw, ew \in \text{External} \quad \text{rcas}(er_1, er_2, erw, ew) \quad r_{\text{val}}(er_1) \neq r_{\text{val}}(erw) \]  
\[ er_1 \xrightarrow{po} er_2 \xrightarrow{po, hb} erw \xrightarrow{po, hb} ew \wedge er_1 \xrightarrow{hb} erw \wedge w_{\text{cov}}(ew) = r_{\text{val}}(erw) \]  
\[ er_1, er_2, erw, ew \in \text{External} \quad \text{rcas}(er_1, er_2, erw, ew) \quad r_{\text{val}}(er_1) = r_{\text{val}}(erw) \]  
\[ er_1 \xrightarrow{po} er_2 \xrightarrow{po, hb} erw \xrightarrow{po, hb} ew \wedge er_1 \xrightarrow{hb} erw \wedge w_{\text{cov}}(ew) = r_{\text{val}}(erw) \wedge r_{\text{val}}(erw) = r_{\text{val}}(er_2) \]  

Figure 4: Axiomatic semantics of the coreRMA language.

remote put and remote get statements. These predicates are true iff their arguments are the actions generated from a remote put or get; the first argument identifies the external read action and the second argument the external write.  

Rule PG orders by both \( \xrightarrow{po} \) and \( \xrightarrow{hb} \), a remote put/get statement’s external read action \( er \) before the companion external write \( erw \). The value read by the external read equals the value written by the external write.  

**Rule for Remote Get Accumulate.** We introduce an analogous rga predicate for remote get accumulate statements.  

Rule GA orders an rga’s external read actions \( er \) before external read-write actions \( erw \) before external write actions \( ew \). The rule also determines the values written: the external read-write value is equal to the sum (or other operation) of the values read by the external read \( er \) and external read write \( erw \) actions. The value written by the external write is equal to the value read by the external read write \( erw \).  

**Rules for Compare and Swap.** We next introduce predicate rcas analogous to \( rp \), \( rg \), and \( rga \). Two rules give the semantics of compare-and-swap: CAS-F for the non-equal case and CAS-T for the equal case. In both cases, the rules order the external read \( er_1 \) before \( er_2 \), both before the external read-write \( erw \), and all before the external write \( ew \). As for the values, the remote compare and swap always writes to \( erw \) the same value as read from \( erw \). If the value read from \( er \) differs from that read from \( erw \), CAS-F gives no further constraints. When the values are equal, the statement writes to \( erw \) the value read from \( er_2 \).  

**Rule for Local Action Ordering.** Rule LO defines the happens-before relation \( \xrightarrow{hb} \) for local actions to include all program order \( \xrightarrow{po} \) relations between local actions.  

**Rule for Write Sequentiality.** Rule WS totally orders, in \( \xrightarrow{hb} \), atomic actions that write to the same memory location. This rule does not apply for non-atomic actions.  

**Local CPU Memory Model.** As described, coreRMA focuses on RMA networks with multiple nodes, where each node (x86, ARM, etc.) executes a single thread. If a CPU with some memory model (e.g., x86, ARM) executes multiple threads, then the current rules applying to local actions (rules F1, LO, R1, R2) must be parametrized with that memory model—currently these rules assume that lo-
6. RMA Validation Framework

We next describe the implementation of our validation framework. The main goal of this framework is to ensure confidence in our formal model, that is, that our model accurately captures the behavior of real-world networks. Based on the formal model, our system automatically generates test cases and then executes these test cases on actual networks, in the process checking for discrepancies between the two. Concretely, we check for two types of suspicious behaviors: (i) violations: behaviors produced by the actual network which contradict what our model (and the official documentation) allows, and (ii) unobserved behaviors: we look for behaviors expected by our model which never appear across multiple executions of the test on the network. As we will see later in Section 7, both of these build confidence that our model accurately captures reality.

6.1 Automatic Test Case Generation

Figure 5 presents the flow of our test case generation framework. The flow consists of the following steps: (i) we first express our formal model in the Alloy Analyzer (Jackson 2006); (ii) we exhaustively generate instances verifying the rules of the model up to a given bound (provided a priori; correlated with the maximum length of the test) and we convert these instances into an intermediate representation; (iii) we compute all possible expected outputs (values for local registers) for each instance; and (iv) we translate the intermediate representation to RMA programs and we execute these instances on real-world networks obtaining the actual outputs. Based on these actual outputs and the expected outputs, we can identify both unexpected behaviors which should not be possible as well as expected outputs which do not seem to occur on the network. We next discuss these steps in more detail.

Defining an Alloy Model. We started our evaluation by encoding the axiomatic semantics of our formal model in Alloy, a lightweight declarative modeling language. The Alloy Analyzer accepts Alloy models and automatically produces satisfying instances using a SAT solver. Our encoding was straightforward and includes around 600 lines of Alloy. The encoding mirrors the semantics of Section 5.

Instance Generation. Having encoded our model in Alloy, we sought to produce instances that illustrate differences between coreRMA and the actual networks (both unexpected and unobserved outputs). We thus queried Alloy for a complete set of model instances, up to a given bound. An instance comprises a set of processes, each with actions, registers, and memory locations. Additionally, Alloy provides the values read and written by the actions. We generate a test body, in our intermediate language, from each instance.

To generate tests that have behaviors forbidden by coreRMA, we ask that the instance generated by Alloy has a cycle in its $\rightarrow^b$ order. To reduce the number of possible tests and increase the efficiency of the search for violations, we require that the $\rightarrow^b$ cycle contains an edge induced by a rule in Figure 4 and that removing this edge renders $\rightarrow$ aacyclic. We call this edge a pivot. We generate tests successively using each rule of Figure 4 as a pivot. Note that tests generated using a certain pivot are guaranteed to exercise the corresponding rule. These tests may also exercise additional rules, depending on the statements that they contain.

To generate tests with behaviors that are easier to observe, we require that each test must contain at least one local read (stored in a unique register) from each Writer action, and that local writes should have distinct values.

We exhaustively generate instances up to a given bound on instance size. The instance size represents the number of actions (actions belong to statements).

Instance generation required a fairly standard implementation of test generation and compiler techniques. We programmatically call Alloy via its API, exhaustively enumerate and extract instances, and produce concise intermediate representation code summarizing each instance. We de-duplicate IR instances because: (i) Alloy is known to generate duplicate instances, and (ii) some instances differed but generated identical intermediate representations.

Figure 6 illustrates a test generated by our tool; the pivot is the $\rightarrow^b$ edge implied by IR between the external write to $x$ generated by $\text{put}(x^a, y)$ and the external read from $x$ generated by $y = \text{get}(x^b)$. The expected outputs represent all expected values of registers $a$ and $b$ according to coreRMA.

Computing All Expected Outputs. For each generated test, we use our coreRMA model to compute all expected outputs. Using the model rules and the given test as constraints, we can query Alloy for one set of expected local register values. Next, we add a negation of that newly obtained output as an additional constraint, and repeat the query until Alloy exhausts all sets of values. Having a complete set of expected outputs allows us to detect, at runtime, unexpected observed outputs as well as unobserved expected outputs.

Code Generation. Finally, we translated each test (written in the intermediate representation) into C. Targets used were the Cray DMAPP API, InfiniBand’s IBV Verbs and Portals 4 API. We note that writing code generators which
Figure 5: *coreRMA* validation: the procedure used to check the accuracy of our model against real-world RMA networks.

\[
\begin{align*}
X &= 1 \\
P0: & \\
a &= X \\
p(X^0, Y) \\
Y &= get(X^0) \\
flush(0) \\
b &= Y
\end{align*}
\]

Expected outputs: \( \langle a, b \rangle \in \{ \langle 0, 0 \rangle, \langle 1, 2 \rangle \} \).

Figure 6: Test generated automatically with IR rule as pivot. The size of the test is 9 and it has 2 processes. The test has two possible outputs according to the *coreRMA* semantics.

would leverage the given APIs was a non-trivial challenge. For instance, it is significantly harder than writing generators for processor memory models; this is because it requires a deep understanding of network implementation details. RMA APIs are unforgiving of errors and would often cryptically refuse to proceed. Establishing the correct setup required several weeks of work. Also, to increase the number of observed behaviors, we introduced small pseudorandom delays between statements in our generated programs. To our knowledge, ours is the first work to comprehensively stress-test RMA networks by generating all instances up to a given bound.

7. Evaluation

In this section, we describe an extensive experimental evaluation for validating our formal model against real-world networks. We considered the following networks:

- **DMAPP** (S-2446-5202) API running on Cray Aries (Faannes et al. 2012) hardware using the Cray x86 compiler. Cray Aries offers parametric in-order routing, and we enable it by using the `DMAPP_ROUTING_DETERMINISTIC` attribute.
- **IBV Verbs API** running on InfiniBand (The InfiniBand Trade Association 2004) hardware, which does not provide strict in-order guarantees for accesses between the same source-destination pair.
- **Portals 4 API**, not backed by hardware, configured to run over UDP (also supports shared memory and OFED).

We generated 7,441 tests. Out of these tests, 3,654 have two interacting processes. By generating tests with two processes, we exercise both the interaction between nodes and the intra-node interactions between the CPU and the NIC. Generating tests with more than two processes would further stress test the interactions between nodes, which might reveal more interesting behaviours.

Depending on the size of the test program, the generation of one test takes between 0.01 and 0.1 seconds. When exhaustively generating tests, Alloy frequently generates multiple instances of the same test, hence total times may increase by a factor of 1000. Overall, generating tests took about 20 hours, and determining all possible outputs of the tests approximately 5 hours.

We executed each test \(10^4\) times and recorded the outputs. Test execution (including connection setup) takes up to 20 seconds per \(10^4\) iterations. (We ran many tests \(10^5\) times and found no additional outputs.) Our tests found network behaviors contradicting existing documentation. Additionally, we did not observe some outputs predicted by our *coreRMA* model—further investigation revealed that the networks provided additional undocumented guarantees.

Table 6 summarizes the results from tests and outputs generated from the stock *coreRMA* rules—these are the rules presented in Section 5.3. When generating tests, we pick a rule (first column) and remove the \(hb\) edge corresponding to a single application of that rule. The second column (# proc) shows the number of processes (1–2) for that row. Single-process tests exercise RMA due to asynchronous interaction between the CPU and the NIC. The third column (Size) shows the bound on the number of actions for each rule. These sizes yielded enough tests of enough complexity such that we could explore the behavior of RMA networks, identifying both unexpected behaviors and unobserved expected outputs. Of course, test generation for still larger sizes is possible using our procedure, limited only by machine availability. For rules \(R1\), \(R2\), and \(LO\), despite the lower bound, the number of tests generated (fourth column—# tests) is high, mainly because these rules do not require the existence of remote statements (which generate additional actions) in the test. The fifth column (ms/test) shows the mean time it takes
Alloy to generate a test. The sixth column (# outputs) indicates coreRMA’s predicted total number of outputs for the tests in that row. The average number of outputs per test is 2.6 (1 output: 3,187 tests; 2 outputs: 1,334; 3 outputs: 910; 4 outputs: 812; 5 outputs: 594; and 6–19: 604 tests). The following columns present results from Cray DMAPP, IBV Verbs, and Portals 4. We indicate the number of tests with outputs contradicting coreRMA predictions (# Errors) and the percentage of observed outputs relative to expected outputs (Obs.%).

**RQ1: Can coreRMA discover tests contradicting existing documentation?** For Cray DMAPP, 135 tests have unexpected outputs because the in-order routing guarantee given by the DMAPP_ROUTING_DETERMINISTIC parameter is not respected. Figure 6 illustrates one example from our tests. Running this test yields unexpected outputs \( \langle a = 0, b = 1 \rangle \) and \( \langle a = 1, b = 1 \rangle \). When \( b = 1 \), remote read \( y = \text{get}(x^0) \) runs before remote write \( \text{put}(x^0, \ y) \). This contradicts IR (remote actions to the same target are executed in the order in which they are issued). Network experts confirmed that these executions violate the available documentation. We reported concrete specification violations to Cray Inc. which triggered immediate replies and were confirmed.

We customized coreRMA to correctly capture the exact guarantees of VPI Verbs on InfiniBand by removing rule IR and imposing ordering between put - put and get - get. After customization, all the outputs we observe are expected.

We discovered that IR is not enforced on VPI Verbs by using stock coreRMA (all the rules in Figure 4) and detecting 4 tests with unexpected outputs. Network experts confirmed that the outputs were indeed allowed. Figure 8 presents one of these tests, which shows 3 unexpected outputs, all with \( c = 0 \). This output demonstrates that the remote read action of get is executed after the remote write action of put, contradicting IR. Our customized coreRMA correctly generates the 3 previously unexpected outputs for this test as expected outputs. An interesting observation is that stock coreRMA (with IR) gives a higher percentage of observed outputs for IR and PG pivots (97%, 99.3% respectively) than the customized coreRMA (49% for both). Overall, the percentage of observed outputs decreases from 94.5% to 88.8%, showing that it is hard to trigger a put - get reordering in practice.

**RQ2: How precisely does coreRMA model real networks?** In practice, we observe approximately 90% of the expected outputs. The percentage of expected outputs according to coreRMA that are actually observed is influenced by the precision of the model, the capacity to reorder certain actions and by the additional guarantees provided by the networks and not captured in coreRMA.

While some of the unobserved outputs are caused by the fact that it is difficult to trigger the behaviors that produce these outputs by inserting delays (non-triggerable behaviors), other unobserved outputs are due to the fact that the networks provide undocumented additional guarantees. (We added these guarantees to the coreRMA model and saw that the percentage of observed behaviors increased).

**Non-Triggerable Behaviors.** We could not introduce delays in our generated tests between actions making up remote statements (put, get, rga, cas). For instance, for the tests generated using as pivot rule GA, we observe 52.7% of the expected outputs for tests with 1 process and 80.4% for tests with 2 processes on Cray DMAPP. We strengthened rule GA to execute the component actions of an rga atomically; in that case, our observations would cover 90% of the newly expected behaviors for 1-process tests and 85% for two processes, without introducing any unexpected outputs. However, this strengthening is not guaranteed by the documentation and should not be assumed when writing code.

**Additional Guarantees.** When running the test shown in Figure 9 on Cray DMAPP, the expected output where all the local variables are equal to 2 is never observed. This output would require that the read action of the put statement is executed after the \( y = 2 \) statement. We discovered that the doc-
Table 6: Tests generated from stock coreRMA semantics identify 148 issues over 7,441 tests.

<table>
<thead>
<tr>
<th>Rule</th>
<th># proc</th>
<th>Size</th>
<th># tests (ms/test)</th>
<th># outputs</th>
<th>Cray DMAPP</th>
<th>VPI Verbs</th>
<th>Portals 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td># errors</td>
<td>Obs (%)</td>
<td># errors</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td># errors</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>1</td>
<td>8</td>
<td>863</td>
<td>20</td>
<td>1492</td>
<td>77.8</td>
<td>67.4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>9</td>
<td>1040</td>
<td>25</td>
<td>3616</td>
<td>99.0</td>
<td>99.7</td>
</tr>
<tr>
<td>R2</td>
<td>1</td>
<td>7</td>
<td>419</td>
<td>15</td>
<td>532</td>
<td>95.4</td>
<td>92.4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8</td>
<td>684</td>
<td>22</td>
<td>1514</td>
<td>96.5</td>
<td>100</td>
</tr>
<tr>
<td>F1</td>
<td>1</td>
<td>9</td>
<td>326</td>
<td>34</td>
<td>338</td>
<td>98.2</td>
<td>96.4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>10</td>
<td>172</td>
<td>49</td>
<td>452</td>
<td>96.4</td>
<td>100</td>
</tr>
<tr>
<td>F2</td>
<td>1</td>
<td>9</td>
<td>350</td>
<td>35</td>
<td>374</td>
<td>96.7</td>
<td>93.5</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>10</td>
<td>188</td>
<td>47</td>
<td>492</td>
<td>94.6</td>
<td>100</td>
</tr>
<tr>
<td>F3</td>
<td>1</td>
<td>10</td>
<td>810</td>
<td>49</td>
<td>1320</td>
<td>96.8</td>
<td>96.8</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>200</td>
<td>52</td>
<td>356</td>
<td>97.7</td>
<td>100</td>
</tr>
<tr>
<td>IR</td>
<td>1</td>
<td>10</td>
<td>104</td>
<td>55</td>
<td>104</td>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>127</td>
<td>120</td>
<td>368</td>
<td>69</td>
<td>96.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GA</td>
<td>1</td>
<td>10</td>
<td>299</td>
<td>27</td>
<td>1091</td>
<td>52.7</td>
<td>52.1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>48</td>
<td>84</td>
<td>624</td>
<td>80.4</td>
<td>75.6</td>
</tr>
<tr>
<td>PG</td>
<td>1</td>
<td>8</td>
<td>164</td>
<td>15</td>
<td>276</td>
<td>2</td>
<td>94.5</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>10</td>
<td>260</td>
<td>27</td>
<td>1024</td>
<td>58</td>
<td>97.1</td>
</tr>
<tr>
<td>CAS-F</td>
<td>1</td>
<td>10</td>
<td>12</td>
<td>36</td>
<td>12</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>48</td>
<td>75</td>
<td>288</td>
<td>88.1</td>
<td>85.4</td>
</tr>
<tr>
<td>CAS-T</td>
<td>1</td>
<td>10</td>
<td>12</td>
<td>36</td>
<td>12</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>48</td>
<td>75</td>
<td>288</td>
<td>90.9</td>
<td>84.3</td>
</tr>
<tr>
<td>LO</td>
<td>1</td>
<td>7</td>
<td>146</td>
<td>21</td>
<td>250</td>
<td>90.4</td>
<td>84.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8</td>
<td>362</td>
<td>19</td>
<td>1004</td>
<td>94.8</td>
<td>100</td>
</tr>
<tr>
<td>WS</td>
<td>1</td>
<td>9</td>
<td>282</td>
<td>34</td>
<td>934</td>
<td>73.3</td>
<td>73.9</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>10</td>
<td>477</td>
<td>47</td>
<td>3176</td>
<td>92.4</td>
<td>90.2</td>
</tr>
</tbody>
</table>

Summary: 7441 tests, 19937 errors, 135 observed errors, 89.9% observed, 94.5% observed, 13 errors, 89.7% observed.

1 Behavior forbidden in stock coreRMA but not contradicting specification.

Documentation stated that, for efficiency reasons, the `put` can directly send to the NIC the data to be written remotely, instead of programming the NIC DMA engine. This happens only for data smaller than a certain threshold, which for DMAPP is 4KB by default. We modeled this additional guarantee in coreRMA, and the observed output percentage for IR 2-process tests increased from 96.4% to 99.7%. This shows that stock coreRMA can be easily customized to match exactly the properties of a specific RMA network. We chose not to add this constraint to coreRMA because it is specific to Cray DMAPP and to the chosen threshold.

In summary, coreRMA has shown value in uncovering behaviors on real world networks not described by existing documentation. The axiomatic rules are precise, 90% of the outputs expected by coreRMA being observed on concrete networks. coreRMA is both easily customizable such that it fits precisely the network guarantees, and general enough to describe the common RMA behaviors, refined throughout our interactions with network experts.

Our framework allows experimentation with finding platform-specific specifications: one may easily add or remove rules, generate tests, and verify whether the hardware conforms to the stated rules. An interesting future work item is to completely automate the process: one could imagine defining a space of rules and automatically selecting those rules which are consistent with experimental results.

8. Related Work

We discuss two kinds of related work: work on analyzing RMA-style programs (e.g., MPI) and general work in the analysis of weak memory models.
Remote Memory Access (RMA) Programming. Specification of RMA libraries and language models is ongoing. The OFED low-level communication interface is currently undergoing a major reform (Hefty 2014). High-level RMA languages have also not stabilized. A new version of MPI RMA is under development. Proposed features such as remote notification (Belli and Hoefler 2015) interact intricately with the memory model.

MPI-3 RMA semantics have been informally described by Hoefler et al. (Hoefler et al. 2013). Detailed semantics for about 200 of 300 MPI-2 API calls are described in TLA+ syntax by Li et al. (Li et al. 2011). Both works show that it is feasible to encode semantics of real-world RMA languages and show the benefits of rigorous specification uncovering minor inconsistencies in the standard text. Our work focuses on a core set of RMA semantics (covering at least MPI-3.0, UPC, and Fortran 2008). This allows us to abstract away from detailed MPI semantics and focus on the core difficulties of RMA programming.


The works closest to us focus on formalizing specific memory models, typically evaluating them with litmus tests (Blanchette et al. 2011; Torlak et al. 2010). The primary focus of these works is to enable language designers to explore non-intuitive behaviors on hand-crafted small examples. For instance, Sarkar et al. (Sarkar et al. 2011) formalize appropriate semantics for Power. Their approach iterates over an a priori fixed set of tests and explores all executions of a test. An important early work in this direction was CheckFence (Burckhardt et al. 2007), which encoded relaxed memory model effects into SAT and demonstrated the feasibility of checking non-trivial concurrent programs.

To our knowledge, we are the first to bounded-exhaustively generate tests for memory models. Although Alglave et al (Alglave et al. 2014b) generate tests using diy, they simply use predefined cycles in the $hb\xrightarrow{\text{ordering}}$ ordering. Also, they model shared-memory, uniform access multiprocessor systems, so they need not distinguish local and remote actions nor atomic and non-atomic actions. Our work, by contrast, generates all examples up to a given bound on a richer model.

A popular line of work focuses on techniques for bounded checking of given programs for models including x86 TSO, PSO, RMO, and Power (Burckhardt and Musuvathi 2008; Kuperstein et al. 2010; Burnim et al. 2011; Liu et al. 2012; Linden and Wolper 2013; Bouajjani et al. 2013; Alglave et al. 2013; Norris and Demsky 2013). There has also been work on infinite-state automatic verification and synchronization synthesis, usually in the form of fences (Kuperstein et al. 2011; Abdulla et al. 2012; Dan et al. 2013; Meshman et al. 2014; Alglave et al. 2014a; Dan et al. 2015). Our work is largely orthogonal: we take the necessary first step of formalizing the memory model itself, which enables verification of RMA programs.

In-order routing is a key feature of RMA, provided by vendors, which distinguishes our work on RMA from related work on asynchronous actions such as X10 AP-GAS (Saraswat et al. 2010). Considering remote statements (put, get, etc.) as asynchronous calls yields no way to enforce in-order routing between a certain action (the remote action) of a first asynchronous call and a remote action from a subsequent asynchronous call, yet at the same time leaving unordered the local actions of the two asynchronous calls. Ignoring in-order routing, asynchronous calls can model remote statements (but the model would also have to handle atomicity).

The work on RMA languages includes dynamic data race and conflict detection for UPC (Park et al. 2011, 2013) and MPI-3 (Chen et al. 2014). The dynamic analysis approaches used in that work therefore use the semantics informally encoded in the RMA language implementation. None of these works provide formal memory model semantics for the programs they consider. Our work formalizes the semantics and enables the development of tools that work independent of implementations. Further, we use constraint solvers, not dynamic analysis or conflict detection, allowing us to check arbitrary safety properties.

9. Conclusion

We introduced the first core calculus, coreRMA, and its axiomatic semantics, to cleanly capture characteristics of Remote Memory Access (RMA) programming. We generated bounded-exhaustive test suites using constraint solvers based on our formal model and tested them on real networks. Our suites revealed actual behaviors which network experts did not expect and showed discrepancies between network behaviors and their documentation. Our work serves as a basis for future work on reasoning about RMA programs and can help troubleshoot and design network implementations.

Acknowledgements

We thank the Swiss National Supercomputing Center (CSCS) for providing us access to Cray Gemini/Aries and InfiniBand networks, Viktor Ivanov for the prototype experiments using DMAPP and VPI Verbs and Larry Kaplan from Cray for the explanation of the unexpected behaviors observed during the evaluation. We also thank Derek Rayside for help with Alloy and Charles Giefer (Intel) for comments on an earlier version of this paper.
References


