Automatic Verification of RMA Programs via Abstraction Extrapolation

Cedric Baumann[1], Andrei Marian Dan[1], Yuri Meshman[2], Torsten Hoefler[1], and Martin Vechev[1]

[1] Department of Computer Science, ETH Zurich, Switzerland
[2] IMDEA Software Institute, Madrid, Spain

Abstract. Remote Memory Access (RMA) networks are emerging as a promising basis for building performant large-scale systems such as MapReduce, scientific computing applications, and others. To achieve this performance, RMA networks exhibit relaxed memory consistency. This means the developer now must manually ensure that the additional relaxed behaviors are not harmful to their application – a task known to be difficult and error-prone. In this paper, we present a method and a system that can automatically address this task. Our approach consists of two ingredients: (i) a *reduction* where we reduce the task of verifying program P running on RMA to the problem of verifying a program $\overline{P}$ on sequential consistency (where $\overline{P}$ captures the required RMA behaviors), and (ii) *abstraction extrapolation*: a new method to automatically discover both, predicates (via *predicate extrapolation*) and abstract transformers (via *boolean program extrapolation*) for $\overline{P}$. This enables us to automatically *extrapolate* the proof of P under sequential consistency (SC) to a proof of P under RMA. We implemented our method and showed it to be effective in automatically verifying, for the first time, several challenging concurrent algorithms under RMA.

1 Introduction

Remote Memory Access (RMA) programming is a technology used in modern data centers to communicate between machines with low overhead. It enables low latencies ($< 1\mu s$ [26]) and high bandwidth. In RMA, remote operations are executed by the underlying network interface controller bypassing the CPU and the operating system (in contrast to regular network operations). The network card reads the data using Direct Memory Access (DMA), sends it over the network, and finally the receiving network card writes the data using DMA. This approach is faster than traditional network communication because in data centers, the direct access propagation delay is small compared to a network message stack overhead of standard sockets. RMA technology is available in several networks: InfiniBand [43], IBM Blue Gene Q [6], IBM PERCS [9], Cray Gemini [7] and Aries [25]. Typically, the RMA functionality is available through RMA libraries (InfiniBands OFED [37], Cray DMAPP [18], Portals4 [12]). Middleware applications, such as the Hadoop File System [32], then call the RMA interface directly.

Newer developments introduce RMA extensions for Ethernet (RoCE [13]) or IP routable RMA protocols (iWARP [40]).

RMA instructions of a program are executed asynchronously, i.e., the execution of the program proceeds without waiting for the completion of the remote RMA operations. To offer guarantees on the completion of remote operations, RMA provides the `flush` statement, which enforces that all remote operations to a certain machine are executed before the execution continues.

As expected, verifying programs running under RMA is challenging because they exhibit additional relaxed behaviors beyond those allowed by sequential consistency (SC). Moreover, programs executing under RMA exhibit behaviors not possible under other relaxed memory models such as Total Store Order (x86 TSO) or Partial Store Order (PSO) [19]. The goal of this paper is to address this challenge, namely, develop automated techniques for verifying RMA programs.

The problem. Given a program P running on an RMA network and a safety specification S, our goal is to answer whether P satisfies S under RMA, indicated as $P \models_{RMA} S$.

Our Work. In our work we approach this challenge via predicate abstraction [28], a method shown effective in verifying concurrent programs [24, 30] and x86 TSO and PSO programs [20]. Standard predicate abstraction ([11, 28]) assumes sequential consistency (SC). Given a program P and a set of predicates $V = \{p_1, \ldots, p_n\}$ over the variables in P, standard predicate abstraction builds a boolean program $B = \mathcal{BP}(P, V)$ that contains one boolean variable for each predicate in V. The boolean program comes with the guarantee that if B satisfies a property S, then the program P satisfies S as well:

$$B \models_{SC} S \implies P \models_{SC} S$$

Checking whether $B \models_{SC} S$ is typically done via a (three-valued) model checker. If a spurious counter-example execution trace is found by the model checker, then the initial set of predicates V is refined and the procedure is repeated. An overview of this approach is illustrated in the left part of Figure 1.

Reduction. However, the above guarantee does not hold when replacing SC with relaxed memory models, such as x86 TSO or PSO, because naive application of standard predicate abstraction for relaxed memory model programs is unsound ([20]). To address this issue, we reduce the problem of verifying RMA programs to SC verification (the Reduction box in Figure 1) by automatically constructing a new program $\overline{P}$ that captures RMA behaviors as part of the program. $\overline{P}$ uses set variables and boolean flags to account for these behaviors. If $\overline{P}$ satisfies a property S under sequential consistency, then P satisfies S under RMA:

$$\overline{P} \models_{SC} S \implies P \models_{RMA} S$$

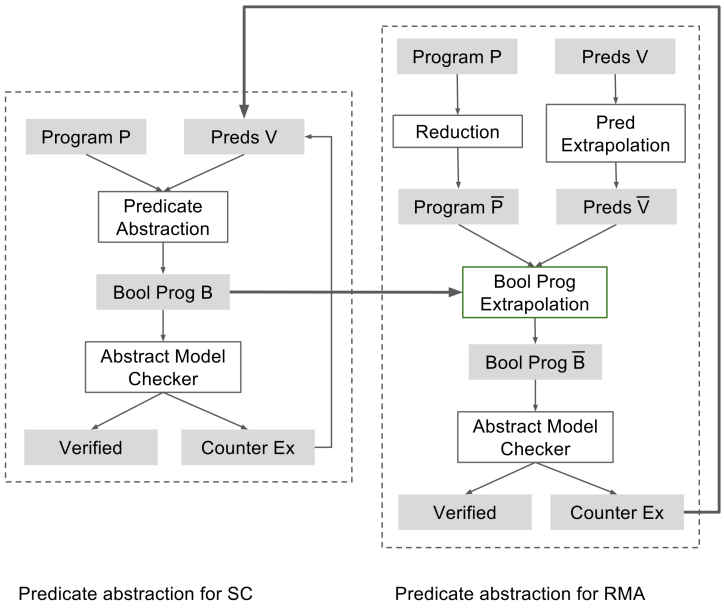

Fig. 1. Predicate abstraction for sequential consistency (left) and our verification method for RMA programs (right).

Predicate extrapolation. Given the newly generated program $\overline{P}$, we automatically generate a set of predicates $\overline{V}$ based on the original set of predicates V (the Pred Extrapolation box in Figure 1). This process is called *predicate extrapolation* and we denote it as $\mathcal{EP}(V) = \overline{V}$. A part of the new predicates in $\overline{V}$ contain universal quantifiers over elements of sets. Compared to the original set of predicates V, the extrapolated set $\overline{V}$ is approximatively twice as large (experimentally observed on our benchmarks), therefore limiting the applicability of standard predicate abstraction (requires an exponential number of calls to an SMT solver, in the worst case). We address this limitation by introducing the *boolean program extrapolation* technique.

Boolean program extrapolation. To side-step the potential exponential blow up, we construct a sound approximation of $\mathcal{BP}(\overline{\mathcal{P}}, \overline{\mathcal{V}})$: we introduce a novel extrapolation function $\mathcal{EBP}$ (the Bool Prog Extrapolation box in Figure 1) to construct a boolean program $\overline{B} = \mathcal{EBP}(\overline{P}, \overline{V}, B)$ without any call to the SMT solver. The *boolean program extrapolation* is based on the boolean program $B = \mathcal{BP}(P, V)$ and satisfies:

$$\overline{B} \models_{SC} S \implies \mathcal{BP}(\overline{P}, \overline{V}) \models_{SC} S$$

Overall approach. Automated verification of $P \models_{RMA} S$ takes place in four steps:

1. *Verify under SC*: given a set of predicates V, build a boolean program $B = \mathcal{BP}(P, V)$ and show that $B \models_{SC} S$.
2. *Reduce to SC*: Build the program $\overline{P}$ (to capture RMA behaviors for P) and extrapolate the corresponding set of predicates $\overline{V} = \mathcal{EP}(V)$ from the predicates V that worked under SC.
3. *Construct extrapolated boolean program*: given program $\overline{P}$, set of predicates $\overline{V}$, and boolean program $B = \mathcal{BP}(P, V)$, construct extrapolated boolean program $\overline{B} = \mathcal{EBP}(\overline{P}, \overline{V}, B)$.
4. *Verify boolean program*: whether $\overline{B} \models_{SC} S$, using a model checker. If the check fails due to abstraction imprecision, we refine the abstraction and restart the verification process. Otherwise, the program P satisfies property S under RMA and the process completes successfully.

Main Contributions. The main contributions of this paper are:

- A program reduction, based on sets, from RMA to SC. The construction allows us to handle traces with infinite number of relaxed memory operations.
- A novel abstraction approach for programs running on RMA, using predicates over sets and a *boolean program extrapolation* technique which requires no calls to the underlying SMT solver for building the boolean program.
- An implementation of our approach in a tool that can, for the first time, automatically verify several challenging (including infinite-state) concurrent algorithms running on the RMA model.

Our work can be seen as a step towards the more general problem of adapting the proof of one program to the proof of a more refined program, technically achieved here via abstraction extrapolation (in the case of predicate abstraction, extrapolation of the boolean program).

2 Overview

We illustrate our approach on a small RMA program shown in Figure 2. In this example, `Process` 1 declares a shared variable Y with initial value 1. `Process` 2 declares shared variables R and X, initializes them to 0 and 2, respectively. Next, it declares a local variable r. It then remotely puts the value of X into Y. Then, X is set to 3 and the value of Y is remotely read from `Process` 1 and is written to R. Finally, the process assigns R to local variable r. The specification (`assert`) is that at the end of the program (`final`), r is different than 3.

Under SC semantics, the only possible value of r at the end of the program is 2. However, under RMA semantics, r can have any value from the set {0,1,2,3}. Note that under other relaxed consistency models, such as x86 TSO, PSO, and C++ RMM, the value of 3 is not possible for r. Yet, under RMA, the `put` from the shared location X to Y, can be delayed until after assigning 3 to X. That value can then be read into R and put into local variable r, leading to an assertion violation. A developer will then have to manually infer the `flush` statements that are required to enforce the specification, when running under RMA semantics. We next introduce the semantics of RMA networks.

2.1 RMA Semantics

In RMA programs, a process can access shared variables of remote processes using remote statements such as `put` or `get`. These remote statements are executed *asynchronously* — the process executing them does not wait for the completion of the remote statement, instead it continues the execution of its program. In hardware, the program executing on a CPU relegates the remote operation execution to a component called Network Interface Controller (NIC) which connects to a NIC on a remote machine. The two NICs complete the operation, on shared locations assigned to the operation, without involving either the local or the remote CPU. `flush` statements are the main mechanism to guarantee that pending remote statements to a specific remote process are completed. A `flush`(pid) statement acts like a barrier, blocking the execution on the process until all pending remote statements to process pid are completed. The `flush` is expensive (increases latency) and should be used sparingly.

```
   Process 1:
1  shared Y = 1;

   Process 2:
1  shared R = 0, X = 2;
2  local r;
3  put(Y, 1, X);
4  store X = 3;
5  R = get(Y, 1);
6  load r = R;

   assert final (r != 3);
```

Fig. 2. RMA program consisting of two processes and a specification (shown at the end) which checks whether the value of local variable `r` is equal to 3.

Syntax. We consider a basic programming language, shown in Table 1, that offers RMA primitives such as `put`, `get` and `flush`. The `load` u = X; statement reads the value of shared variable X and writes it in local variable u. The `store` X = expr statement writes to shared variable X the value of the expression on the right hand side (arithmetic expression over local variables). The `put` and `get` statements operate over shared variables X and Y, and also take as argument the identifier of the process storing the remote variable. The `flush` statement takes as argument a process id. The semantics of these statements are described next.

Table 1. Basic statements which capture the essence of RMA programming.

Statement	Description
`load` u = X;	local read
`store` X = expr;	local write
X = `get`(Y, pid);	remote get
`put`(Y, pid, X);	remote put
`flush`(pid)	flush

Semantics. Let *Procs* be a finite set of process identifiers and $p \in Procs$ a process id. Let *Vars* be the set of all variables. We assume that each variable is uniquely identified (no two variables have the same name) and we define $proc : Vars \to Procs$ as the function mapping each variable to the process where it is declared. We define a transition system as a tuple (s_0, Σ, T), where Σ is the set of program states, $s_0 \in \Sigma$ is the initial state, and $T \subseteq \Sigma \times Actions \times \Sigma$ is a transition relation. The *Actions* set contains all statements in the simple language, and the nic_r and nic_w actions which correspond to the asynchronous

non-deterministic execution of the remote statements:

$$Actions = \{put, get, nic_r, nic_w, load, store, flush\}$$

A program state $s \in \Sigma$ is a tuple $\langle pc, M, R, W \rangle$, where:

- $pc : Procs \to Labels$ is the map from process identifiers to labels. The next label that comes after label $l \in Labels$ is denoted by $n(l)$.
- $M : Vars \to D$ is the state of the memory, mapping each variable to a value in the domain D.
- R is a mapping from process ids to the set of pending remote read operations triggered by the given process. For $p \in Procs$, each read operation $r \in R(p)$ has a variable to be read, denoted with $var(r) \in Vars$. Additionally, r is mapped to a following write action denoted $succ(r)$.
- W is the mapping from process identifiers to the set of pending remote write operations triggered by the given process. Given $p \in Procs$, for each $w \in W(p)$, we define the variable to be written, $var(w) \in Vars$ and the value to be written $val(w) \in D$.

The initial state s_0 has both, the set of pending reads and the set of pending writes initialized to be empty ($\forall p \in Procs : R(p) = W(p) = \emptyset$). These semantics are introduced by [17] and match the configuration without in-order routing, where operations are not ordered between the same source and destination processes. Each rule corresponds to a transition $s \xrightarrow{a} s'$, where $a \in Actions$ and $s, s' \in \Sigma$, $s = \langle pc, M, R, W \rangle$ and $s' = \langle pc', M', R', W' \rangle$.

When a *put* action is executed, a pending read operation r is added to R, and a following write operation w is declared. The variables read and written by r and w correspond to the arguments of the `put` statement. The *get* action has similar semantics. The execution of the pending read operations in R is non-deterministic and, after reading the value of the target variable, the following write operation is added to the set of pending writes W. Similarly, the pending writes are executed non-deterministically. The local *store* and *load* actions are executed synchronously and interact directly with the memory M (storing a value, or reading from memory), without using the pending operation sets R and W. Assuming process $p \in Procs$ issues a *flush* action, after its execution the set $W(p)$ does not contain any pending write operations to the target process of the flush. Similarly, the set $R(p)$ does not contain any pending read operations from the target process of the flush, and additionally none of the successor write operations of the pending read operations in $R(p)$ write to the target process. Next, we briefly recap standard predicate abstraction, assuming SC.

2.2 Predicate Abstraction under Sequential Consistency

This section illustrates the standard predicate abstraction procedure applied to the example program in Figure 2, assuming SC.

SC semantics. When restricting an RMA program to SC, we assume that all remote operations (e.g., the `get` statements in Figure 2) are executed synchronously. For example, the R = `get`(Y, 1) statement has the same semantics as a normal assignment R = Y. For our example, the predicates sufficient to verify the specification are:

$$V = \{(r \neq 3), (R \neq 3), (Y \neq 3), (X \neq 3)\}$$

This set of predicates can be determined either manually or using a counter example guided refinement loop. The result of applying predicate abstraction on the example program using the predicates in V is a concurrent boolean program that soundly represents all possible behaviors of the original program.

Boolean program construction. The resulting boolean program has four boolean variables $\{B_1, B_2, B_3, B_4\}$ (one for each predicate in V). For each statement of the program P, standard predicate abstraction computes how the statement changes the values of the predicates. For instance, statement R = `get`(Y, 1) (which for SC we interpret as R = Y) at line 5 in `Process` 2 assigns to variable B_2 (corresponding to the predicate $(R \neq 3)$) the value of B_3. We say that the predicate B_2 is updated using the *cube* of size 1 containing the predicate B_3. Intuitively, $(R \neq 3)$ holds after the statement R = `get`(Y, 1) if $(Y \neq 3)$ holds before the statement. More details about standard predicate abstraction are presented in subsection 4.1.

The complexity of building the boolean program using standard predicate abstraction is exponential in the number of predicates in V ([11, 28]) and its main component is the search of *cubes* (conjunctions of predicates or negated predicates that imply a given formula). Optimizations such as bounding the size of cubes to a constant k lead to a complexity of $|V|^k$, by building a coarser abstraction, therefore losing precision.

2.3 Predicate Abstraction for RMA Programs

We next illustrate our RMA verification approach which is based on *extrapolating* the proof of the program under sequential consistency (discussed in more detail in subsection 2.2).

Step 1: Verify P under SC. The input for this step is the program P (Figure 2) and the set of predicates V shown in subsection 2.2. Here, we assume all remote statements are executed *synchronously*. After the program is verified, the resulting boolean program $B = \mathcal{BP}(P, V)$ will be used for extrapolation in the third step of our approach.

Step 2: Construct the reduced program $\overline{P}$. We reduce the problem of verifying P under RMA semantics to the problem of verifying a new program $\overline{P}$ under SC. The program $\overline{P}$ non-deterministically accounts for all possible asynchronous behaviors of P under RMA.

Auxiliary variables. To construct $\overline{P}$, we introduce auxiliary variables of two types: sets and boolean flags. Additionally, we use two methods: addToSet, that takes as arguments a set and an element, and adds this element to the set, and randomElem, that takes as input a set and returns a random element of the set. The sets accumulate all values that can be read by a `get` statement, or all possible values that can be written remotely by a `put` statement. In our example, we introduce two set variables: X1Set and Y1Set. For example, variable X1Set stores all values of variable X that the `put` statement at line 5 in Figure 2 can write to Y. Initially, X1Set and Y1Set are empty. A boolean flag is introduced for each remote statement. It represents whether the remote statement is pending to be executed asynchronously by the underlying network. For the example in Figure 2, we introduce variables Put1Active and Get1Active, initially set to `false`, corresponding to the `put` and `get` statements.

Statement translation. Next, for each statement of P, we generate a corresponding code in $\overline{P}$. The result of translating the program in Figure 2 is partly shown in Figure 3 (only translation for lines $1-4$ of the original program is shown).

```
   Process 2:
1  shared R = 0, X = 2,
2  Put1Active = false,
3  Get1Active = false,
4  Y1Set = ∅,
5  X1Set = ∅;
6  local r;
7  // put(Y, 1, X);
8  if( !Put1Active )
9    Put1Active = true;
10   X1Set = {X}
11 else
12   addToSet(X1Set,X);
13 // X=3;
14 if(Put1Active && ⋆)
15   Y = randomElem(X1Set);
16   addToSet(Y1Set,Y);
17   if(⋆) Put1Active=false;
18 store X = 3;
19 if( Put1Active )
20     addToSet(X1Set,X);
21 ⋮
```

Fig. 3. Running example translation excerpt of $\overline{P}$: this program contains the RMA behaviors affecting the original program P.

Lines $2-5$ initialize the introduced auxiliary variables (initializing boolean flags such as Put1Active and sets such as X1Set). Lines $7-12$ represent the translation of the `put` statement, where the flag Put1Active is set to `true` and the current value of variable X is added to X1Set. Next, the $X = 3$ statement of program P corresponds to lines 13-20. If the `put` operation is active, then the value 3 is added to X1Set (in line 20). Before it, remote operations are non-deterministically executed. Lines 14-17 represent the non-deterministic asynchronous execution of a pending remote operation. In this case, the only pending operation corresponds to the `put` statement. The $\star$ in the condition of line 14 represents a non-deterministic choice of whether to take the branch or not. Line 15 selects a random element from the set X1Set and assigns it to Y. The transformation process is described in detail in subsection 3.1.

Predicate extrapolation. For the newly generated program $\overline{P}$ shown in Figure 3, our technique automatically extrapolates the set of predicates V to a new set of

predicates $\overline{V} = \mathcal{EP}(V)$. Given the newly introduced set variables, we generate predicates that are universally quantified over all the elements of a set. For example, given the initial SC predicate $(X \neq 3)$, we generate the quantified predicate $\forall e \in X1Set : e \neq 3$. For simplicity, we denote the predicate as $(X1Set \neq 3)$. We assign a special logic to the case where $(X1Set \neq 3)$ is false - it implies that all the elements in $X1Set$ are equal to 3:

$$(X1Set \neq 3) = \begin{cases} true, & \forall e \in X1Set : e \neq 3 \\ false, & \forall e \in X1Set : e = 3 \\ \star, & otherwise \end{cases}$$

This predicate allows to keep track of the values added to the set. When all the elements of the set are different than 3, then the predicate is *true*. *Importantly*, the predicate is *false* only when all the elements of the set are equal to 3; otherwise, the value of the predicate is unknown, and we denote this value by $\star$. The result of applying predicate extrapolation will return the following set:

$$\overline{V} = V \cup \{(X1Set \neq 3), (Put1Active = true)\}$$

For each boolean flag that the program translation introduces (e.g., Put1Active), the predicate extrapolation will add a predicate that tracks the value of the flag. In our example, the new predicate is $(Put1Active = true)$. More details about predicate extrapolation are presented in subsection 4.2.

Step 3: Construct the Extrapolated Boolean Program After performing the second step, we obtain a program $\overline{P}$ and a set of predicates $\overline{V}$. Applying standard predicate abstraction and building a boolean program $\mathcal{BP}(\overline{P}, \overline{V})$ requires a significantly higher computational effort than building $\mathcal{BP}(P, V)$. The reason is that $\overline{P}$ contains more instructions than P that have to be analyzed, and $\overline{V}$ has more predicates than V. Instead, we generalize the idea of predicate extrapolation [20] to boolean program extrapolation: starting from the boolean program constructed for the SC semantics $B = \mathcal{BP}(P, V)$, we construct a new boolean program $\overline{B} = \mathcal{EBP}(B, \overline{P}, \overline{V})$. By extrapolating the boolean program, we avoid performing additional cube search (we do not require calls to an SMT solver) to construct $\overline{B}$, because we extrapolate cubes already found for the construction of B.

For instance, at line 15 of Figure 3, the statement Y = randomElem(X1Set), where a random element of X1Set is selected and assigned to Y, the boolean program extrapolation will use as input the transformers in the boolean program B that correspond to the statement `put`(Y, 1, X) from Figure 2. For the statement `put`(Y, 1, X), the predicate $(Y \neq 3)$ is assigned the value of $(X \neq 3)$. We extrapolate this boolean assignment for the statement `put`(Y, 1, X), and the predicate $(Y \neq 3)$ is assigned the value of the predicate $(X1Set \neq 3)$. If the predicate $(X1Set \neq 3)$ is *true*, then all the elements inside the set X1Set are different than 3. Therefore, selecting a random element of the set and assigning it to X makes the value of $(Y \neq 3)$ *true*. If the predicate $(X1Set \neq 3)$ is *false*, it means that the elements in X1Set are equal to 3, and assigning any of them to Y

makes the predicate ($Y \neq 3$) *false*. A formal description of the boolean program extrapolation is presented in subsection 4.3.

Step 4: Model Checking We verify if $\overline{B}$, constructed in Step 3, satisfies the specification S, using a model checker. If there is no reachable state in the program that contradicts S, then the verification succeeds. If an error state is discovered, there are two possibilities: either the error state is spurious, and we refine the abstraction, or it is a valid error state, and the original program P needs more `flush` statements such that property S holds under RMA. If we add `flush` in all possible locations the program is guaranteed to verify, since the program is then limited to SC executions.

Verification results. The specification holds under RMA semantics with a single `flush` statement added to the program, after the `put`(Y, 1, X) instruction of line 3. It is important to observe that this allows the program to retain RMA specific behaviours that do not violate the assertion and are not possible under SC. For example, the value $r = 0$ is not possible under SC, while under RMA, the value of r at the end of the program can be 0. This is because an asynchronous `get` operation, can be executed even after `Process` 2 assigns the value of R to r. Although the `get` reads into R the value 2 from Y (that value reached there with the `put` of line 3 and the following `flush`), the line 6 assignment will write the value of R from initialization into r. Adding a `flush` after the `get` statement would eliminate this state under RMA. However, the state satisfies the specification of interest (r $\neq$ 3), and our procedure successfully identifies which `flush` statement is not required.

3 Reduction of RMA programs

In this section, we describe the source-to-source transformation of a program P running under RMA semantics to a new program $\overline{P}$ running under SC semantics, such that $\overline{P}$ soundly approximates P. The main idea is to generate a program $\overline{P}$ which encodes the RMA semantics of program P.

3.1 Reduction: RMA to SC

We define the translation function that takes as input a statement from program P and returns a list of statements of program $\overline{P}$:

$$[\![]\!] : \mathit{Stmt} \rightarrow \mathit{List}\langle \mathit{Stmt} \rangle$$

Set variables. The newly generated program $\overline{P}$ contains, in addition to the variables of program P, two types of auxiliary variables that contribute to capturing the semantics of RMA programs. Let *sid* be a mapping that takes as input a

Table 2. The source to source translation of statements from program P running on RMA to a new program $\overline{P}$ running on sequential consistency. p is the identifier of the process that executes the statement.

$[\![$`store` X = a$]\!]$	$[\![$`put`(Y, pid, X)$]\!]$
remoteOps(X) `store` X = a; $\triangleright \forall s \in read(X)$: `if` $active(s)$ addToSet($set(s)$,X);	`if` (! $active(s)$) $active(s)$ = `true`; $set(s)$ = {X}; `else` addToSet($set(s)$, X);
$[\![$`flush`(pid)$]\!]$	remoteOps(X)
`while` ($\bigvee_{s \in remote(p,pid)} active(s)$) $\triangleright \forall s \in chain(remote(p, pid))$: `if` ($active(s) \wedge \star$) $trg(s)$ = randomElem($set(s)$); `if` ($\star$) $active(s)$ = `false`;	`while` ($\star$) $\triangleright \forall s \in chain(write(X))$: `if` ($active(s) \wedge \star$) $trg(s)$ = randomElem($set(s)$); addToSet($Sets(trg(s))$, $trg(s)$); `if` ($\star$) $active(s)$ = `false`;

remote statement (`put` or `get`) of program P and returns a unique identifier of that statement. The first type of auxiliary variables are sets. For each remote `get` statement s of the form var_dst = `get`(var_src, pid) we introduce a set variable. The name of the set variable is the concatenation of the var_src variable, the unique identifier of the `get` statement $sid(s)$ and the string "Set". For example, assuming the statement X = `get`(Y, pid) with unique identifier 1, we generate the set variable Y1Set. Similarly, for each `put` statement `put`(var_dst, pid, var_src), we generate a set variable corresponding to var_src and the statement identifier. All the set variables are initially empty. Given a remote statement s, let $set(s)$ be the set variable associated to s in our translation. Let $Sets$ be the set of all set variables of the program and $Sets(X)$ the sets corresponding to a variable X.

Boolean flags. The second type of auxiliary variable is a boolean flag. For each remote `put` and `get` statement, we introduce a boolean variable, with a name that is the concatenation of "Get"/"Put", the statement id $sid(s)$ and the string "Active". For instance, given a statement `put`(Y, pid, X) with statement id equal to 3, we add the boolean flag Put3Active. All the auxiliary boolean variables are initially `false`. Let $Flags$ be the set of all boolean flags. The mapping $active : Sets \rightarrow Flags$ returns the boolean variable that corresponds to the same remote statement as the set variable given as argument.

Chains of remote statements. Given two remote statements s_1 and s_2, the relation $s_1 \circ s_2$ holds if s_1 writes to a variable X and s_2 reads or writes X. Let $\circ^+$ be the transitive closure of $\circ$. We define $chain(s_2)$ as the set of remote statements s

such that $s \circ^{+} s_2$. We overload the *chain* operator to sets of remote statements: given S a set of remote statements, $chain(S) = \bigcup_{s \in S} chain(s)$.

Notation. For a given shared variable X, there are potentially several set variables, one for each remote statement that reads the value of X. Let $write(X)$ be the set of remote statements that write to X and $read(X)$ the set of remote statements reading from X. We denote by $remote(p, pid)$ all the statements executed by process p that remotely read or write from process pid. The variable that is written by a remote statement s is $trg(s)$. These functions are used in the source to source translation.

Translation of program statements. Table 2 illustrates the source-to-source translation of program P running on RMA to a new program $\overline{P}$ that runs on SC and captures all the behaviors of P.

Non-deterministic execution of remote operations. Since RMA remote operations are executed asynchronously, they could be executed at any point in the program after the statement that triggers them. An exact translation would non-deterministically execute, in any order, every pending RMA statement at each program point. However, the state space of the resulting program would grow significantly, making the verification more challenging.

In our approach, the resulting program $\overline{P}$ contains, at specific points, code that executes the pending remote operations non-deterministically. This code is described in Table 2 as remoteOps, and is parameterized by a shared variable X. The statements executed non-deterministically are all the remote statements that write to X and all the remote statements that form *chains* with the statements writing to X, denoted $chain(write(X))$. For each statement $s \in chain(write(X))$, if the active flag corresponding to s ($active(s)$) is `true`, then non-deterministically ($\star$) assign to the target variable of s ($trg(s)$) a random element of the set variable corresponding to s ($set(s)$). Next, the active flag is non-deterministically set to `false` (there can be potentially several instances of statement s pending, in case s is executed in a loop).

Local store. The statement `store` X = a writes a local variable a or numerical value to a shared variable X that belongs to the process executing the statement. In the translation, we first add the remoteOps(X) code. Next, we add the value of a to the corresponding sets of the remote statements to all the set variables that correspond to remote statements reading from X (denoted $read(X)$).

Remote Put. A `put`(Y, pid, X) statement reads a shared variable X of the local process and writes its value to the shared variable Y at the remote process pid. This operation is done asynchronously by the underlying network. There are potentially several values that the local read from X can observe, when X is modified by the program, following the remote `put` statement. Similarly, the remote write operation to Y at the remote process pid happens non-deterministically after the read from X.

For our reduction, the statement s = `put`(Y, pid, X) is translated by first checking if the flag variable ($active(s)$) is `false`. In this case, the s becomes active (by setting $active(s)$ =`true`), and we initialize $set(s)$ with the current value of X ($set(s) = \{X\}$). If the $active(s)$ flag is `true` (that can be the case if s is part of a loop), then the current value of X is added to $set(s)$.

Flush. The `flush`(pid) statement makes sure that after its execution all active RMA operations from the current process (p) to the remote process pid (denoted by $remote(p, pid)$) are executed. We translate the `flush` statement as a loop that executes pending operations as long as at least one of the RMA statements in $remote(p, pid)$ is still pending. The loop contains non-deterministic statements to execute each of the pending statements in $remote(p, pid)$ and statements that create *chains* with the pending statements in $remote(p, pid)$. This translation is sound, as it covers all the possible orders of execution of the pending statements.

4 Predicate Abstraction for RMA Programs

This section describes how to adapt predicate abstraction to the task of verifying RMA programs. The key idea is to cheaply build an RMA proof from the SC proof by reusing predicates and transformers of the SC program proof. Note that the safety property that we want to prove remains the same as in the SC case. We begin by describing standard construction of predicate abstraction for SC.

4.1 Predicate Abstraction

Predicate abstraction [11, 28] is a form of abstract interpretation that employs Cartesian abstraction over a set of predicates. Given a program P, and vocabulary (set of predicates) $V = \{p_1, \ldots, p_n\}$ with corresponding boolean variables $\hat{V} = \{b_1, \ldots, b_n\}$, predicate abstraction constructs a boolean program $\mathcal{BP}(P, V)$ that conservatively represents behaviors of P using only boolean variables from $\hat{V}$ (corresponding to predicates in V). We use $[p_i]$ to denote the boolean variable b_i corresponding to p_i. We note that the mapping is a bijection. We similarly extend $[\cdot]$ to any boolean function φ.

Constructing $\mathcal{BP}(P, V)$. A literal is a predicate $p \in V$ or its negation. A *cube* is a conjunction of literals, and the size of a cube is the number of literals it contains. The concrete (symbolic) domain is defined as formulae over all possible predicates. The abstract domain contains all the cubes over the variables $p_i \in V$. Predicate strengthening F_V maps a formula φ from the concrete domain to the largest disjunction of cubes (over V), d, such that $d \Longrightarrow \varphi$. The abstract transformer of a statement st, w.r.t. a given vocabulary V, can be computed using the weakest-precondition ([11]), while performing implication checks using an SMT solver. For each $b_i \in \hat{V}$ the abstract transformer generates:

$$b_i = choose([F_V(wp(st, p_i))], [F_V(wp(st, \neg p_i,))])$$

where $choose(\varphi_t, \varphi_f) = \begin{cases} true, & \varphi_t \text{ evaluates to } true \\ false, & \text{only } \varphi_f \text{ evaluates to } true \\ \star, & \text{otherwise} \end{cases}$

For example, given the predicates $V = \{(X > 0), (Y > 0), (Z > 2)\}$, the corresponding boolean variables $\hat{V} = \{b_1, b_2, b_3\}$, and a statement X = Y + Z, predicate abstraction generates the abstract transformer: $b_1 = choose(b_2 \wedge b_3, false)$. After executing the statement X = Y + Z, $(X > 0)$ holds if $(Y > 0)$ and $(Z > 2)$ hold before the statement, otherwise $(X > 0)$ becomes $\star$. Different predicate abstraction techniques use different heuristics for reducing the number of calls to the prover.

4.2 Predicate Extrapolation, $\overline{V} = \mathcal{EP}(V)$

After the SC predicate abstraction is successfully completed, we *extrapolate* the set V of SC predicates, and we obtain $\overline{V}$, the predicates for the reduced program $\overline{P}$. The set $\overline{V}$ of predicates consists of: (i) the predicates V from the SC proof, (ii) universally quantified predicates for each set variable, based on the extrapolation of SC predicates, and (iii) predicates for the boolean flag variables.

Extrapolation of SC predicates for each set variable. A sound optimization of the source-to-source translation is to generate just one variable *XSet* for each shared variable X, instead of one set variable per remote statement. In the rest of the paper, we denote *XSet* the set corresponding to X. The abstraction accounts for the set variables and tracks predicates that hold for the values contained in these sets. We generate for each predicate $p \in V$, that references a shared variable X, a corresponding predicate for *XSet*, $\forall e \in XSet : p[^e/_X]$. The newly generated predicate contains a universal quantifier over all the elements of the set. We denote this predicate as $p[^{XSet}/_X]$.

Logic of the predicates over set variables. A predicate $\forall e \in XSet : p[^e/_X]$ over a set variable *XSet* is *true* if and only if predicate p is *true* for every element of *XSet*. However, we refine the case where the predicate $\forall e \in XSet : p[^e/_X]$ is *false*: if $p[^e/_X]$ is false for every element $e \in XSet$. Overall, the set predicates have the following logic:

$$p[^{XSet}/_X] = \begin{cases} true & \forall e \in XSet : p[^e/_X] \\ false & \forall e \in XSet : \neg p[^e/_X] \\ \star & otherwise \end{cases}$$

For example, assume we are given *XSet* (initially empty), the predicate $p = (X > 5)$, such that $p[^{XSet}/_X] = (XSet > 5)$ and a sequence of addToSet statements that successively add the values 6, 7 and 4 to *XSet*. After executing the statement addToSet(XSet, 6), the predicate $p[^{XSet}/_X]$ becomes *true*. After addToSet(XSet, 7), the predicate $p[^{XSet}/_X]$ remains *true*. After addToSet(XSet, 4), $p[^{XSet}/_X]$ becomes $\star$, because neither all elements in *XSet* are greater than 5 nor it is the case that all elements of *XSet* are less or equal than 5.

If we select a random element from $XSet$ using T = randomElem(XSet), then the value of a predicate $p[^{XSet}/_X]$ is the same as $p[^e/_X]$, where $e \in XSet$. This can be used to derive the value of predicates that contain variable T. For example, given the predicates $(T > 3)$ and $(XSet > 5)$, if $(XSet > 5)$ is *true* before T = randomElem(XSet), then $(T > 3)$ is *true* after the statement is executed. Similarly, if $(XSet > 5)$ is *false*, then a predicate such as $(T > 5)$ becomes *false* after the statement, by using the special logic we assign to the set predicates.

In our implementation, for predicates that reference two shared variables (e.g., $(X < Y)$) we extrapolate for each shared variable separately and do not generate a predicate involving two set variables (e.g., $(XSet < Y)$ and $(X < YSet)$ are generated, while $(XSet < YSet)$ is not generated). This provides a good trade-off between precision and efficiency. We observe that this over-approximation is precise enough and there is no need to track directly the relation between two set variables. We show in subsection 4.3 how to soundly handle this abstraction.

Generation of predicates for the boolean flags. For each remote operation (for example a `get` operation at label *lbl*) the translation presented in subsection 3.1 generates a boolean flag variable to indicate when the remote operation (or an instance of the operation, in case it is executed in a loop) is pending to be executed (for the `get` operation, a get_lbl_flag boolean variable). For each such boolean variable we generate a corresponding predicate that captures the boolean flag state (e.g., for the `get` operation at label *lbl*, we generate the predicate $(get\_lbl\_flag = 1)$).

4.3 Boolean Program Extrapolation, $\overline{B} = \mathcal{EBP}(\overline{P}, \overline{V}, B)$

Given the extrapolated predicates $\overline{V}$, a standard construction of the boolean program is typically quite expensive, because the number of predicates $|\overline{V}|$ and the size of the program $|\overline{P}|$ are significantly larger than in the SC case. This observation was shown in [20] for relaxed memory models, a work which proposed cube extrapolation to reduce the number of calls to an underlying theorem prover. In our work, in addition to handling a more complex memory model (RMA) and generating more complex predicates that contain quantifiers, we introduce a novel boolean program extrapolation method that builds a boolean program without any calls to the theorem prover. The resulting boolean program $\overline{B}$ is a sound over-approximation of $BP(\overline{P}, \overline{V})$.

Transformers for the set operations. The main difference between programs P and $\overline{P}$ is that the latter contains set variables and statements that operate on the set variables (set initialization, addToSet and randomElem). The predicates in $\overline{V}$ refer to the set variables and we next describe how to compute the transformers for the set operators. We again note that attempting to directly calculate their transformers would require a large number of calls to the theorem prover.

Transformers for set initialization. We construct the translation such that all the set initialization statements have the form XSet = {X}). A set XSet is always

initialized with the singleton set containing the value of the variable X (the variable to which the set corresponds). As shown in subsection 4.2, the predicate extrapolation generates for every predicate $p \in V$ that contains X a predicate $p[^{XSet}/_X]$. Therefore, after executing the statement XSet = {X}, the predicates containing XSet have the same value as the predicates containing X:

$$p[^{XSet}/_X] = p$$

Transformers for addToSet. All addToSet have the form addToSet(XSet, X), adding the value of a variable X to the set XSet that corresponds to that variable. The predicates $p[^{XSet}/_X] \in \overline{V}$ are updated after addToSet(XSet, X) such that:

$$p[^{XSet}/_X] = \mathit{choose}(p[^{XSet}/_X] \wedge p, \neg p[^{XSet}/_X] \wedge \neg p)$$

If all the elements in XSet satisfy the predicate p and the value of X satisfies P, then, after adding the value of X to XSet, all the elements of XSet still satisfy p.

Transformers for randomElem. Every statement Y = randomElem(XSet) in program $\overline{P}$ corresponds to a remote operation such as Y = `get`(X) or `put`(Y, X) from program P. For the SC verification, these remote statements are assumed to be synchronous assignments of the form Y = X. During the SC predicate abstraction, for each predicate $p \in V$ that contains Y, we compute φ_t and φ_f, the disjunctions of cubes that appear as arguments of the *choose* function that updates p: $p = \mathit{choose}(\varphi_t, \varphi_f)$. In the case of the Y = randomElem(XSet) statement, for all predicates $p \in V$ that contain Y, we update them using the formula:

$$p = \mathit{choose}(\varphi_t[^{XSet}/_X], \varphi_f[^{XSet}/_X])$$

Consider an example with $(X > 7), (Y > 5) \in V$ and a statement Y = `get`(X) in P with the SC transformer $(Y > 5) = \mathit{choose}((X > 7), \mathit{false})$. For the corresponding Y = randomElem(XSet) statement in $\overline{P}$, we generate the transformer $(Y > 5) = \mathit{choose}((\mathit{XSet} > 7), \mathit{false})$.

The extrapolated predicates $p \in \overline{V}$ that contain both Y and at least one set variable are updated to $\star$ after the Y = randomElem(XSet) statement. This sound over-approximation is required because $\overline{V}$ has no predicates that contain more than one set variable. For example, given the predicates $(X \geq Z), (Y \geq X), (Y \geq Z) \in V$, the extrapolated predicate $(Y \geq \mathit{ZSet}) \in \overline{V}$, after a statement Y = randomElem(XSet) the predicate $(Y \geq \mathit{ZSet})$ is set to $\star$, as we do not track the predicate $(\mathit{XSet} \geq \mathit{ZSet})$ that is required for a more precise transformer.

5 Experimental Evaluation

We implemented an analysis tool for RMA programs based on the method described so far. In this section, we discuss our experimental evaluation of the tool on a number of challenging concurrent algorithms running on RMA networks. The experiments ran on an Intel(R) Xeon(R) 2.13GHz with 250GB RAM. The

first research question is whether predicate and boolean program extrapolation are sufficiently scalable to verify all benchmarks. The second question deals with the precision of the abstractions we introduced and whether we can compute the smallest required `flush` placement for each program such that our tool is precise enough to prove that the specification holds under RMA.

Benchmarks. We tested our analyzer on 14 challenging concurrent algorithms: Dekker [23], Peterson [38], Szymansky [42] mutual exclusion algorithms, an Alternating Bit Protocol (ABP), an Array-based Lock-Free Queue, Lamport's Bakery algorithm [34], the Ticket locking algorithm [8], the Pc1, Pgsql, Kessel, BlueTooth, Sober, Driver Qw, loop2_TLM programs as defined in [14], and an RMA Lock [41]. The benchmarks have two or three processes and the number of lines of code is between 25 and 85. Several programs have an infinite number of states (ABP, Queue, Bakery, Ticket). The safety properties are either mutual exclusion or reachability invariants involving labels of different processes. For each benchmark, the safety property is the same for both SC and RMA.

5.1 Prototype Implementation

We implemented the RMA analyzer in Java (around $9,000$ lines of code). For the cube search (when building the boolean program for SC verification), the tool uses Z3 [22] as an underlying SMT solver. We use the 3-valued model checker Fender (implemented in Java) to check if the boolean program satisfies the specification. Fender also uses Z3 for abstraction refinement. We made minor changes to the error trace construction and interpolation methods of Fender in order to accommodate the RMA abstraction based on sets.

Flush search. For an input program, we initially add a `flush` statement after each RMA remote statement (`put` or `get`). Alternatively, the user can suggest a different initial `flush` placement. The analyzer starts checking the input program using *all* the flushes of the initial `flush` placement. If the analyzer successfully verifies the program, then the `flush` search process continues by removing one `flush` statement, updating the boolean program and rechecking the property using Fender (no need to rerun the predicate abstraction).

We develop a search procedure for the smallest placement of `flush` statements for which our tool successfully proves that the program satisfies its specification under RMA. We choose a mix between breadth-first and depth-first search. In the first phase (breadth-first search of depth 1), we repeatedly check the program while removing one of the `flush` statements of the initial placement. This phase identifies the `flush` statements that are always needed for the program to satisfy the specification. In the second phase, the tool performs a depth-first search, while trying to remove only `flush` statements that were successfully removed in the first phase. This hybrid solution is much faster than a simple depth-first or breadth-first search, especially for the cases where the number of `flush` statements needed is small. Finally, the search returns one or several solutions of flush placements that make the program satisfy the desired property.

Table 3. Experimental results showing verification of a number of algorithms on both SC and RMA models.

	SC predicate abstraction			RMA predicate abstraction			
Algorithm	$\|V\|$	$\mathcal{BP}(P,V)$ (s)	$B(loc)$	$\|\overline{V}\|$	$\overline{B}$ (loc)	Fender (s)	Min `flush`
Dekker	11	1	498	29	2068	876	4/12
Peterson	10	1	356	21	1045	4	4/7
Abp	16	1	485	20	662	1	1/2
Pc1	18	2	658	35	3797	126	2/7
Pgsql	12	1	418	18	1549	1	2/4
Qw	13	2	487	29	1544	1345	4/5
Sober	23	8	831	48	8466	10	0/9
Kessel	18	3	534	36	1621	16	5/10
Loop2_TLM	29	165	1068	43	1986	3960	4/4
Szymanski	34	228	1182	64	7081	316	7/14
Queue	13	24	572	22	1104	13	1/2
Ticket	17	114	640	43	3615	4320	5/6
Bakery	19	330	828	41	2947	288	6/10
RMA Lock	24	50	763	60	5932	65679	9/18

5.2 Experimental Results

The results of the analysis are presented in Table 3. For the first part of our analysis, we perform the verification of the programs assuming SC.

Meaning of table columns. $|V|$ represents the number of predicates used for SC verification. To obtain these predicates, we started with a manually selected set, then used abstraction refinement to find the sufficient set to verify the program under SC. The $\mathcal{BP}(P,V)$ (s) column records the duration in seconds of building the boolean program abstraction. Most of this time (95%) is spent in the SMT solver, used for the cube search. $B(loc)$ shows the number of lines of code in the resulting boolean program. Checking the SC boolean program with Fender takes for each algorithm a small number of seconds, therefore we omit it from the results table. In the second step of our analysis, we perform the boolean program extrapolation, based on the SC boolean program B. $|\overline{V}|$ is the number of predicates after extrapolating the SC predicates V ($\overline{V} = \mathcal{EP}(V)$). The column $\overline{B}(loc)$ shows the number of lines of code of the extrapolated boolean program $\overline{B}$. The *Fender (s)* column shows the runtime of Fender for checking whether $\overline{B}$ satisfies the specification. Finally, *Min* `flush` is the result of the search for the minimal number of `flush` statements required for the program to satisfy its specification under RMA. The first number is the smallest number of flushes for which the verification succeeds, and the second number is the number of flushes of the initial `flush` placement.

On average, the extrapolated boolean program is 5 times larger than the SC boolean program. The resulting extrapolated predicates are twice as many, on average, compared to the original predicates $|V|$ (note that $V \subset \overline{V}$). We obtain

larger running times for the Ticket and Loop2_TLM benchmarks, due to the high number of predicates and the complexity of the programs.

Scalability of Boolean program extrapolation. The boolean program extrapolation that constructs $\overline{B}$ takes under a second for each benchmark. If we took the approach of using standard predicate abstraction of the reduced program $\overline{P}$, the time would be significantly higher. For instance, we experimented with the Bakery mutual exclusion algorithm, and the standard approach took over three hours (compared to sub-second times for the extrapolation). The precision of the two boolean programs is similar, as the same minimal `flush` placements are found for both. This shows the advantage of our extrapolation method.

Extrapolation precision and minimal `flush` placement. The extrapolated boolean program is precise enough to remove `flush` statements and verify the property for all benchmark algorithms (except Loop2_TLM). For Loop2_TLM, the model checker timed out after two hours, while checking the boolean program with a `flush` removed. Surprisingly, the Sober algorithm does not require any `flush` statement under RMA. This is due to the algorithm already executing the remote operations in loops that have the same effect as a `flush` (by checking in their condition the value returned by the `get` statement). Comparing these `flush` placements with other memory models (x86 TSO, PSO), is challenging, due to the one sided aspect of the remote operations. Two `store` operations to a shared variable X, one in each thread, under TSO, become a `store` and a `put` in the RMA program, since X belongs to on one of the processes.

6 Related Work

Remote Memory Access (RMA) Programming. The semantics of MPI-3 RMA have been first described, informally, by [31]. The work of [17] introduces operational semantics for Partitioned Global Address Spaces (PGAS), which follow the same principles as RMA programs. The focus in their work is analysing robustness of the programs using PGAS. Our work, on the other hand, focuses on proving that safety specifications hold for a program under RMA by using predicate abstraction. Axiomatic semantics of the core functionality of RMA programs are introduced in [19], which shows the benefits of formal specifications in discovering inconsistencies in existing RMA libraries.

Program Analysis under Weak Memory Models There exists significant body of work in automatically verifying programs and synthesizing fences required for the correctness of programs running under relaxed memory models such as x86 TSO, PSO, Power, C11. This is the first work that verifies infinite-state concurrent programs running on RMA. The work closest to ours is [20], that introduces predicate extrapolation and cube extrapolation for verifying programs under PSO and x86 TSO (more restricted than RMA). While cube extrapolation reduces the search space of cubes when constructing the boolean program, in this

work we introduce complete boolean program extrapolation that side-steps cube search while building the abstraction. Another important difference is that, while [20] abstracts only bounded store buffers, in this work we handle unbounded sets of pending operations via sets and quantified predicates. This results in potentially less `flush` operations needed to enforce the specification. The work of [3] defines a general framework for verifying programs under weak memory models, based on the axiomatic semantics. In our work, we rely on operational semantics of RMA for the source-to-source reduction to SC. The work of [2] uses predicate abstraction to verify x86 TSO programs, while discovering predicates using traditional refinement techniques. In our work, based on a more strict semantics (SC), we directly discover the abstraction for weaker semantics (RMA) via extrapolation. Work on predicate abstraction for infinite-state concurrent programs assuming SC and using compositional methods such as Owicki-Gries and rely-guarantee is presented in [29, 30].

Reduction to SC The reduction of verifying programs under weak memory models to verification under SC via program transformation is also used in [5, 10, 21, 36]. This work introduces a new transformation and abstraction for RMA programs, that is precise enough to verify the program specifications while using a reduced number of `flush` statements. Works by [15, 16, 33, 35] consider verification of finite-state programs under weak memory models, considering just some of the sources of infinite-state programs (e.g. unbounded store buffers or infinite variable domains). Infinite-state programs are handled in [4] for x86 TSO. In recent work, [1] explores the advantages of alternative semantics for TSO (replacing store buffers with load buffers) that is more efficiently verified. In the reduction step of our work, the auxiliary set variables resemble load buffers, because when a remote write operation is performed, the value to be written is selected randomly from the set, which collects all values that the corresponding remote read operation might have. [39] introduces a procedure that detects unexpected executions that might occur when porting the program from a source to a target memory consistency model.

7 Conclusion

We introduced the first automatic verification technique for programs running on RMA networks. The key idea is abstraction *extrapolation*: automatically build an abstraction of the program for a relaxed memory model such as RMA, based on an existing abstraction of the program under SC. We implemented the predicate and boolean extrapolation methods and we successfully verified several challenging concurrent algorithms running on RMA. To our knowledge, this the first time these programs have been verified on the RMA memory model. We believe this work takes a step towards applying proof extrapolation techniques to other hardware or software relaxed memory consistency models.

References

1. Abdulla, P. A., Atig, M. F., Bouajjani, A., and Ngo, T. P. The benefits of duality in verifying concurrent programs under TSO. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada* (2016), J. Desharnais and R. Jagadeesan, Eds., vol. 59 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 5:1–5:15.
2. Abdulla, P. A., Atig, M. F., Chen, Y.-F., Leonardsson, C., and Rezine, A. Automatic fence insertion in integer programs via predicate abstraction. In *Proceedings of the 19th International Conference on Static Analysis* (Berlin, Heidelberg, 2012), SAS'12, Springer-Verlag, pp. 164–180.
3. Alglave, J., and Cousot, P. Ogre and pythia: an invariance proof method for weak consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017* (2017), G. Castagna and A. D. Gordon, Eds., ACM, pp. 3–18.
4. Alglave, J., Kroening, D., Nimal, V., and Poetzl, D. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings* (2014), A. Biere and R. Bloem, Eds., vol. 8559 of *Lecture Notes in Computer Science*, Springer, pp. 508–524.
5. Alglave, J., Kroening, D., Nimal, V., and Tautschnig, M. *Software Verification for Weak Memory via Program Transformation.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 512–532.
6. Allen, F., Almasi, G., Andreoni, W., Beece, D., Berne, B. J., Bright, A., Brunheroto, J., Cascaval, C., Castanos, J., Coteus, P., Crumley, P., Curioni, A., Denneau, M., Donath, W., Eleftheriou, M., Fitch, B., Fleischer, B., Georgiou, C. J., Germain, R., Giampapa, M., Gresh, D., Gupta, M., Haring, R., Ho, H., Hochschild, P., Hummel, S., Jonas, T., Lieber, D., Martyna, G., Maturu, K., Moreira, J., Newns, D., Newton, M., Philhower, R., Picunko, T., Pitera, J., Pitman, M., Rand, R., Royyuru, A., Salapura, V., Sanomiya, A., Shah, R., Sham, Y., Singh, S., Snir, M., Suits, F., Swetz, R., Swope, W. C., Vishnumurthy, N., Ward, T. J. C., Warren, H., and Zhou, R. Blue Gene: A vision for protein science using a petaflop supercomputer. *IBM Syst. J. 40*, 2 (Feb. 2001), 310–327.
7. Alverson, R., Roweth, D., and Kaplan, L. The Gemini system interconnect. In *Proc. of the IEEE Symposium on High Performance Interconnects (HOTI'10)* (2010), IEEE Computer Society, pp. 83–87.
8. Andrews, G. R. *Concurrent programming - principles and practice.* Benjamin/Cummings, 1991.
9. Arimilli, B., Arimilli, R., Chung, V., Clark, S., Denzel, W., Drerup, B., Hoefler, T., Joyner, J., Lewis, J., Li, J., Ni, N., and Rajamony, R. The PERCS high-performance interconnect. In *Proc. of the IEEE Symposium on High Performance Interconnects (HOTI'10)* (Aug. 2010), IEEE Computer Society, pp. 75–82.
10. Atig, M. F., Bouajjani, A., and Parlato, G. *Getting Rid of Store-Buffers in TSO Analysis.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 99–115.
11. Ball, T., Majumdar, R., Millstein, T. D., and Rajamani, S. K. Automatic predicate abstraction of C programs. In *Proceedings of the 2001 ACM SIGPLAN*

Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001 (2001), M. Burke and M. L. Soffa, Eds., ACM, pp. 203–213.

12. Barrett, B. W., Brightwell, R. B., Pedretti, K. T. T., Wheeler, K. B., Hemmert, K. S., Riesen, R. E., Underwood, K. D., Maccabe, A. B., and Hudson, T. B. The Portals 4.0 network programming interface. Tech. rep., Sandia National Laboratories, 2012. SAND2012-10087.
13. Beck, M., and Kagan, M. Performance evaluation of the RDMA over Ethernet (RoCE) standard in enterprise data centers infrastructure. In *Proc. of the Workshop on Data Center - Converged and Virtual Ethernet Switching (DC-CaVES'11)* (2011), ITCP, pp. 9–15.
14. Bertrand Jeannet. The ConcurInterproc Analyzer, Sept 2017. available at: `http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi` (Sept. 2017).
15. Bouajjani, A., Derevenetc, E., and Meyer, R. Checking and enforcing robustness against TSO. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings* (2013), M. Felleisen and P. Gardner, Eds., vol. 7792 of *Lecture Notes in Computer Science*, Springer, pp. 533–553.
16. Burckhardt, S., and Musuvathi, M. Effective program verification for relaxed memory models. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings* (2008), A. Gupta and S. Malik, Eds., vol. 5123 of *Lecture Notes in Computer Science*, Springer, pp. 107–120.
17. Calin, G., Derevenetc, E., Majumdar, R., and Meyer, R. A theory of partitioned global address spaces. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2013, December 12-14, 2013, Guwahati, India* (2013), A. Seth and N. K. Vishnoi, Eds., vol. 24 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 127–139.
18. Cray Inc. Using the GNI and DMAPP APIs. Ver. S-2446-52, March 2014. available at: `http://docs.cray.com/` (Mar. 2014).
19. Dan, A. M., Lam, P., Hoefler, T., and Vechev, M. T. Modeling and analysis of remote memory access programming. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016* (2016), E. Visser and Y. Smaragdakis, Eds., ACM, pp. 129–144.
20. Dan, A. M., Meshman, Y., Vechev, M. T., and Yahav, E. Predicate abstraction for relaxed memory models. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings* (2013), F. Logozzo and M. Fähndrich, Eds., vol. 7935 of *Lecture Notes in Computer Science*, Springer, pp. 84–104.
21. Dan, A. M., Meshman, Y., Vechev, M. T., and Yahav, E. Effective abstractions for verification under relaxed memory models. In *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings* (2015), D. D'Souza, A. Lal, and K. G. Larsen, Eds., vol. 8931 of *Lecture Notes in Computer Science*, Springer, pp. 449–466.

22. de Moura, L. M., and Bjørner, N. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings* (2008), C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963 of *Lecture Notes in Computer Science*, Springer, pp. 337–340.
23. Dijkstra, E. Cooperating sequential processes, TR EWD-123. Tech. rep., Technological University, Eindhoven, 1965.
24. Donaldson, A. F., Kaiser, A., Kroening, D., and Wahl, T. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In Gopalakrishnan and Qadeer [27], pp. 356–371.
25. Faanes, G., Bataineh, A., Roweth, D., Court, T., Froese, E., Alverson, B., Johnson, T., Kopnick, J., Higgins, M., and Reinhard, J. Cray Cascade: A scalable HPC system based on a Dragonfly network. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)* (2012), IEEE Computer Society, pp. 103:1–103:9.
26. Gerstenberger, R., Besta, M., and Hoefler, T. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proc. of the ACM/IEEE Supercomputing* (2013), SC '13, pp. 53:1–53:12.
27. Gopalakrishnan, G., and Qadeer, S., Eds. *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings* (2011), vol. 6806 of *Lecture Notes in Computer Science*, Springer.
28. Graf, S., and Saïdi, H. Construction of abstract state graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings* (1997), O. Grumberg, Ed., vol. 1254 of *Lecture Notes in Computer Science*, Springer, pp. 72–83.
29. Gupta, A., Popeea, C., and Rybalchenko, A. Predicate abstraction and refinement for verifying multi-threaded programs. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011* (2011), T. Ball and M. Sagiv, Eds., ACM, pp. 331–344.
30. Gupta, A., Popeea, C., and Rybalchenko, A. Threader: A constraint-based verifier for multi-threaded programs. In Gopalakrishnan and Qadeer [27], pp. 412–417.
31. Hoefler, T., Dinan, J., Thakur, R., Barrett, B., Balaji, P., Gropp, W., and Underwood, K. Remote Memory Access Programming in MPI-3. *ACM Transactions on Parallel Computing (TOPC)* (Jan. 2015).
32. Islam, N. S., Rahman, M. W., Jose, J., Rajachandrasekar, R., Wang, H., Subramoni, H., Murthy, C., and Panda, D. K. High performance RDMA-based design of HDFS over InfiniBand. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 35:1–35:35.
33. Kuperstein, M., Vechev, M. T., and Yahav, E. Partial-coherence abstractions for relaxed memory models. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011* (2011), M. W. Hall and D. A. Padua, Eds., ACM, pp. 187–198.
34. Lamport, L. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM 17*, 8 (1974), 453–455.

35. Linden, A., and Wolper, P. An automata-based symbolic approach for verifying programs on relaxed memory models. In *Model Checking Software - 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010. Proceedings* (2010), J. van de Pol and M. Weber, Eds., vol. 6349 of *Lecture Notes in Computer Science*, Springer, pp. 212–226.
36. Meshman, Y., Dan, A. M., Vechev, M. T., and Yahav, E. Synthesis of memory fences via refinement propagation. In *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings* (2014), M. Müller-Olm and H. Seidl, Eds., vol. 8723 of *Lecture Notes in Computer Science*, Springer, pp. 237–252.
37. OpenFabrics Alliance (OFA). OpenFabrics Enterprise Distribution (OFED) www.openfabrics.org, 2014.
38. Peterson, G. L. Myths about the mutual exclusion problem. *Inf. Process. Lett. 12*, 3 (1981), 115–116.
39. Ponce de León, H., Furbach, F., Heljanko, K., and Meyer, R. Portability analysis for axiomatic memory models. PORTHOS: one tool for all models. *CoRR abs/1702.06704* (2017).
40. Recio, R., Metzler, B., Culley, P., Hilland, J., and Garcia, D. A Remote Direct Memory Access Protocol Specification. RFC 5040, RFC Editor, October 2007.
41. Schmid, P., Besta, M., and Hoefler, T. High-Performance Distributed RMA Locks. In *Proceedings of the 25th Symposium on High-Performance Parallel and Distributed Computing (HPDC'16)* (Jun. 2016).
42. Szymanski, B. K. A simple solution to lamport's concurrent programming problem with linear wait. In *International Conference on Supercomputing* (1988), pp. 621–626.
43. The InfiniBand Trade Association. *Infiniband Architecture Spec. Vol. 1, Rel. 1.2*. InfiniBand Trade Association, 2004.