

# Rethinking Data Race Detection in RMA-Analyzer

Radjasouria Vinayagame \*

Université de Bordeaux,  
Centre Inria de l'université de Bordeaux, Atos  
38130 Echirolles - France  
radjasouria.vinayagame@atos.net

---

## Abstract

Supercomputers are capable of more and more computations, and nodes forming them need to communicate even more efficiently with each other. Thus, other types of communication models gain traction in the community. For instance, the Message Passing Interface (MPI) proposes a communication model based on one-sided communications called the MPI Remote Memory Access (MPI-RMA). Thanks to these operations, applications can improve the overlap of communications with computations. However, one-sided communications are complex to write since they are subject to data races. Tools trying to help developers by providing a data race detection for one-sided programs are thus emerging. This paper rethinks an existing data race detection algorithm for MPI-RMA programs by improving the way it stores memory accesses, thus improving its accuracy and reducing the overhead at runtime.

**Mots-clés :** HPC, MPI-RMA, Consistence mémoire, Analyse dynamique, Binary Search Tree

---

## 1. Introduction

To meet the exascale challenge, parallel programming models are evolving and tend towards programming models that abstract the machine nature with task-based programming or by exposing the hardware at the software level (network with RMA capabilities) with the Partitioned Global Address Space (PGAS) model. The PGAS communication model is based on one-sided communications which decouple data movement from synchronization.

The Message Passing Interface (MPI) standard proposes a similar communication model via MPI Remote Memory Access (MPI-RMA). With MPI-RMA, each MPI process makes a part of its memory available to other MPI processes so the latter can remotely read and write on this “distributed shared memory” with fewer synchronizations than two-sided MPI communications. Consequently, applications migrating from two-sided MPI to MPI-RMA operations should get significant speedup. Nevertheless, most MPI programs still use two-sided communications because MPI-RMA programs are challenging to write and error prone, especially because of data races. Indeed, like with shared memory models, developers must ensure memory consistency in MPI-RMA programs. Thus, some tools aim at helping developers write correct and productive MPI-RMA applications to promote overlap and asynchrony. However, these tools imply a noteworthy overhead at runtime induced by the analysis of the program and are

---

\*. The paper had been proofread by Emmanuelle Saillard, Samuel Thibault, Antoine Capra, Pierre Lemarinier, Van Man Nguyen and Marc Sergent.

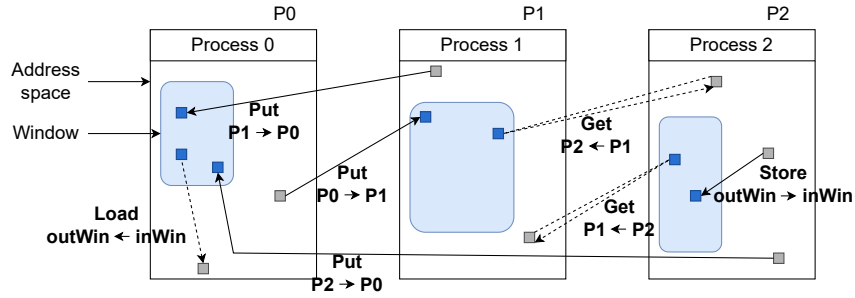


FIGURE 1 – Presentation of possible access operations within an epoch.

not scalable. This paper enhances an existing on-the-fly data race detection tool for MPI-RMA programs called RMA-Analyzer [1]. We propose a new insertion in a Binary Search Tree (BST) algorithm that enhances its scalability and reduces the number of false positives.

In Section 2, we provide some background elements and the key-concepts of MPI-RMA this work relies on. In Section 3, we list the existing tools to detect data races in MPI-RMA programs. Afterwards, in Section 4, we propose a new algorithm that enhances the memory accesses management in RMA-Analyzer in order to reduce the dynamic analysis time. In this section, we also discuss the potential of the tool and future works. Finally, Section 5 concludes this work.

## 2. Background

The third major release of the Message Passing Interface (MPI-3) [12] enhances the one-sided communications presented in the MPI-2 standard and proposes more synchronization operations for the Passive Target mode. An overview of communication options in the MPI-3 specification for RMA operations is presented by Hoeffler et al. in [6]. In this paper, we focus on the Passive Target synchronization mode, in which only the origin process is involved in the synchronization, since it is the closest mode to the synchronization model presented by the PGAS model.

### 2.1. MPI-RMA

MPI-RMA allows each MPI process to have a "distributed shared memory" that can be remotely accessed by other MPI processes. In MPI-RMA, these memory regions are called *windows* and remote memory accesses to these windows are possible during an *epoch*.

Within an epoch, MPI-RMA proposes several communication operations which involve two processes : the origin process which issues the MPI-RMA communication, and the target process whose window is accessed via the communication. In this paper, we focus on two asynchronous one-sided communication operations : *Put* which allows to write a value owned by the origin to the window of the target and *Get* which allows the origin to locally retrieve a value from the window of the target. An example of all possible accesses within an epoch is shown in Figure 1.

When using MPI-RMA, four types of accesses should be considered [1] depending on if the operation is local to the process (Local\_\*) or if it is a remote access (RMA\_\*) and if the operation is a WRITE operation (\*\_Write) or a READ operation (\*\_Read). For instance, a *Put* operation is an RMA\_Write for target and an RMA\_Read for origin. In the same way, a *Get* operation is an RMA\_Read for target and an RMA\_Write for origin. A *Store* operation is a Local\_Write

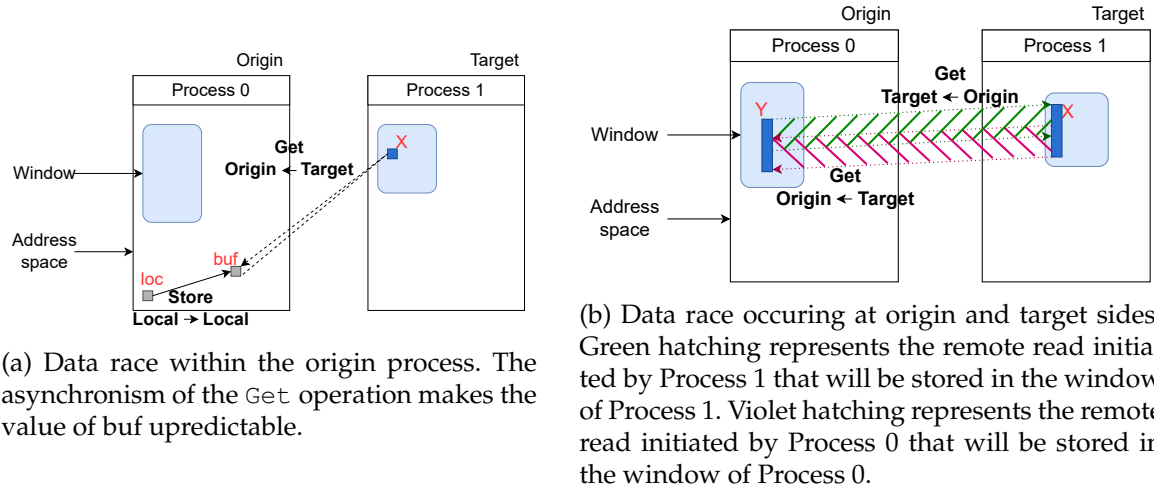


FIGURE 2 – Examples of a data races within a process (a) and between processes (b).

for the process using it.

MPI-RMA ensures three properties that allow an efficient overlap of communications with computations. The first property is completion : since MPI-RMA communications are asynchronous, we cannot know if a communication has completed until the end of the epoch. The second property is ordering : MPI-RMA communications can happen in any order within an epoch. Finally, the third property is atomicity : the atomicity of MPI-RMA communications is only guaranteed at the MPI\_Datatype level.

MPI-RMA has been successfully used by some applications that have migrated from MPI two-sided to MPI-RMA. For instance, *Mizan-RMA* [10], which is a graph processing framework, got a speedup up to 280% using MPI-RMA compared to a previous version that uses MPI\_Send and MPI\_Recv operations. Similarly, the *Graph500* [11] data intensive benchmark got a speedup of 200%. The authors claim that these speedups are achieved through a better overlapping of communications and computations.

## 2.2. Data Races in MPI-RMA Programs

MPI-RMA programs expose memory through an abstraction that allows to read from and write to distant memory at any time, which can lead to data races. A data race occurs if two operations access the same memory range (at least one of them being an RMA access) and at least one of them is a WRITE access (RMA\_Write or Local\_Write). As a consequence, the result may change depending on the execution. A more precise description of data races is presented by Hoefler et al. in [6]. Two examples of data races are presented in Figure 2. In Figure 2a, origin issues a Get operation to store in buf a value owned by target. At the same time, origin writes over buf by making a store operation (buf=loc). Therefore, a data race can occur at the origin side and buf will take the value of either X or loc. Figure 2b shows data races at both origin and target. This is due to the fact that both processes are retrieving the remote value in their own window. That is why, depending on where the retrieved remote value is stored (in the window or out of the window), a data race may occur.

It should be noted that data races can also occur with more than two processes, for instance in the case where multiple processes issue a Put operation to the same space of the same target. All possible cases of data races are reported in Figure 5 in Appendix.

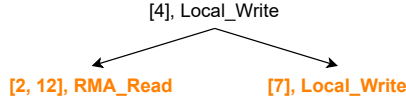
### 3. Related Work

Tools detecting data races for multithreading programs exist (e.g., Valgrind DRD or Helgrind). However, these tools can not be directly applied to MPI-RMA programs since the latter has a different semantic than multithreading programs. There are few approaches to detect data races in MPI-RMA programs. Park et al. [13] present an approach that creates a mirror window each time a window is created. Then, each time a new MPI-RMA communication accesses a memory space in the window, a check for data races is performed in the corresponding mirror window containing previous accesses to that window. This approach does not consider local Load and Store accesses, which leads to false negatives. Moreover, its implementation is only compatible with the MPI-2 standard. *MC-Checker* [3] uses a post-mortem analysis based on a Directed Acyclic Graph (DAG) to detect concurrent regions (in respect to the happens-before relation [8]). *MC-Checker* has been enhanced with *MC-CChecker* [4] which reduces the number of false positives and improves its scalability through a clock-based approach based on the encoded vector clock. Nonetheless, the implementation proposed by the authors of *MC-CChecker* is only compatible with the MPI-2 standard and thus does not support the new MPI-RMA features (such as `MPI_Win_lock_all`/`MPI_Win_unlock_all` functions). *MUST-RMA* [15] is an on-the-fly data race detector for MPI-RMA programs which combines *MUST* [5] and *Thread-Sanitizer*, a shared-memory data race detector [16]. *MUST-RMA* constructs (using the happens-before relation) concurrent regions and "transfers" them to *Thread-Sanitizer* to check for data races. A static analysis detecting data races in MPI-RMA programs has been proposed by Saillard et al. [14]. This analysis works by making a Breadth First Search (BFS) on the Control Flow Graph (CFG) to detect at compile time data races at the origin side only.

*RMA-Analyzer* [1] captures memory accesses during the execution of programs and stores them in a Binary Search Tree (BST). A memory access includes information about the exact interval of addresses that are accessed (we are only considering consecutive accesses which means that all the addresses in the interval are accessed), the type of the access, and debug information (e.g. the location of the access in the source code). Thus, as soon as a data race is detected, *RMA-Analyzer* stops the program and returns an error message including debug information to facilitate the correction of the program. When an MPI window is created, each MPI process creates a BST storing all the memory accesses associated to the addresses it owns (in the window and locally). In its reference implementation, when a new access is issued, a first traversal of the BST is done in order to check for data races with the existing accesses. If there is no error, another traversal is done to insert the new access. This approach has been recently integrated in *PARCOACH* [7], a tool dedicated to the detection of deadlocks caused by improper use of collective calls in parallel applications. We argue that the reference implementation of the approach has performance bottlenecks especially because of the size of the BST that is equal to the number of accesses. Moreover, false negatives are possible because of the approximation made by only considering the lower bound of the interval of addresses to compare the accesses between them. These limitations are due to the fact that accesses present in the BST are not disjoint (leads to false negatives) and are not merged (leads to slowdown). To solve these issues, we introduce a new insertion algorithm that increases the accuracy and reduces the memory footprint and analysis time of the data race detection proposed by *RMA-Analyzer*.

*MC-Checker*, *MC-CChecker* and *RMA-Analyzer* implementations have a lightweight static support that reduces the number of Load/Store instrumentations. A summary of all existing tools is presented in Figure 6 in the appendix.

1. LOAD(4) P0  
2. Put(2, 12) P0  
3. STORE(7) P0



(a) False negative with reference implementation.

Instruction in P0:  
LOAD(4)  
Put(2, 12)  
STORE(7)



(b) False negative detected with our implementation.

FIGURE 3 – Built BST with the reference implementation (left side) and our implementation (right side). Bold accesses are conflicting.

#### 4. New Data Race Detection

Our new data race algorithm improves the insertion of memory accesses in the BST. This new algorithm is based on two main parts : the fragmentation of the accesses (presented in Section 4.1) and the merging of the accesses (presented in Section 4.2). More details about the algorithm are presented in Algorithm 1 in the appendix and an example of its execution is presented in Figure 10 in the appendix.

##### 4.1. Improving the Accuracy of the Insertion Algorithm

The reference implementation can have false negatives caused by non-disjoint accesses. Indeed, the accesses are ordered based on their lower bound, therefore the insertion of a new access in the BST may miss a conflict with another access and potentially miss a data race because of the approximation made by considering only the lower bound of the interval of addresses to compare the accesses between them (the accesses are then intersecting). An example of a false negative is presented in Figure 3a. In this example, the memory interval corresponding at the Put ([2...12], RMA\_Read) will be inserted in the left subtree of the node ([4], Local\_Write). Thus, when a new access ([7], Local\_Write) is inserted in the BST, it is inserted in the right subtree of ([4], Local\_Write) without noticing the intersection between ([2...12], RMA\_Read) and [7], which is a data race.

To make the accesses disjoint, we propose a new fragmentation algorithm that is called each time a new access is inserted in the BST. More details are presented in Algorithm 2 in the appendix. Using this new algorithm, the BST of the reference implementation presented in Figure 3a becomes the BST presented in Figure 3b.

When a new access `new_acc` is inserted in the BST, if this `new_acc` is intersecting with another access `BST_acc` already present in the BST, three new accesses (representing the parts of the intersection of the two intervals) are created and inserted in the BST if they are not empty. The first access represents the leftmost subsection, that is not part of the intersection between `new_acc` and `BST_acc`. This `l_acc` is necessary in order to keep debug information of the latest access to addresses it represents to be finer in error reporting. The second access represents the intersection of `new_acc` and `BST_acc`. Given the access type of `new_acc` and `BST_acc`, the access type of the resulting access `intersection_acc` is given in Figure 7 in the appendix. Basically, RMA accesses prevail on local accesses and WRITE accesses prevail on READ accesses. The third access `r_acc` represents the remaining subsection. An illustration of the fragmentation algorithm on two accesses is shown in Figure 8 in the appendix.

This new fragmentation algorithm has been implemented in PARCOACH and applied on the

P0 (Origin)	P1 (Target)	P0 (Origin)	P1 (Target)
	<i>Window location X</i>		<i>Window location X</i>
Win_lock_all	Win_lock_all	Win_lock_all	Win_lock_all
temp = buf[4]		<b>for</b> (i=0; i<1000; i++)	
<b>Put(buf[2], 10, X)</b>		<b>Get</b> (buf[i], 1, X)	
<b>buf[7] = 1234</b>		<b>Get</b> (buf[0], 1, X)	
Win_unlock_all	Win_unlock_all	Win_unlock_all	Win_unlock_all

(a) False negative program.

(b) Merge loop program.

FIGURE 4 – Comparison codes.

program shown in Figure 4a. It successfully detected the data race between the bold statements, whereas the reference version of the detection algorithm failed to raise the error.

However, this fragmentation algorithm can lead to a drastic increase of accesses representations in the BST. Indeed, each new access possibly increases the number of accesses in the BST by two (one access is removed and three accesses are added). This increase of nodes in the BST may lead to an explosion of the memory usage and slow down the operations on the BST.

#### 4.2. Optimizing the Number of Nodes

In order to avoid the explosion of the number of nodes in the BST, which could negatively impact the scalability of the data race detection algorithm, we propose a node-merging algorithm. To merge two accesses, two conditions need to be ensured. The accesses have to be intersecting or adjacent, and must have the same access type. Thus, each time a new access Acc is inserted in the BST, our algorithm goes through all the accesses that are intersecting with Acc or adjacent to Acc, and merges them with Acc (cf Algorithm 3 in the appendix). Figure 9 in the appendix illustrates the execution of the merging algorithm.

For instance, in the case of the program presented in Figure 4b that has multiple adjacent memory accesses (5,002 memory accesses are made in this program), the reference implementation induces a number of nodes in the BST that is linear in the number of iterations (1,000 in this case). The merging algorithm merges all the nodes induced by the *Get* communications into only one node in the BST. Thus, from a BST that has 5,002 nodes in the BST using the reference implementation, each iteration adding five new nodes to the BST (the remaining two nodes are the first LOCAL\_READ from *i*=0 and the last RMA\_WRITE from *Get* (buf[0], 1, X)), our implementation gets a BST of size two : one node for the variable *i* and one node for all the *Get* accesses.

If we look at the complexity of the algorithm, since the most computationally intensive operations used in the new insertion algorithm are researches, insertions and deletions which are logarithmic in time as we use a (balanced) BST, our new insertion algorithm is also logarithmic in time.

#### 4.3. Discussion and validation

With our new insertion algorithm that is more accurate than the reference algorithm due to the fragmentation of the accesses, and the reduction of nodes in the BST induced by the merging algorithm, we argue that we have contributed to an improved scalability of the overall analysis. Indeed, since the execution time of insertion and deletion operations in the BST depends on the size of the BST, a reduction of its size would also reduce the execution time of the analysis. We let for future work an in-depth analysis of the gain obtained by our implementation. It should be noticed that the approach proposed in this paper can be applied to other programming

languages or libraries such as UPC++ [2]. It is noteworthy that the improvement proposed in this paper has been integrated to PARCOACH which is open-source<sup>2</sup>. The current version of PARCOACH supports only memory addresses that are consecutives, but this approach can be extended to cases where accesses are not adjacent (e.g., in the case of an array of structures in which only parts of each element of the array have to be sent).

To validate our approach, the tool has been tested on a suite of microbenchmark. These tests showed that the new implementation is less prone to false negative than the reference implementation.

## 5. Conclusion

In this paper, we propose a better data race detection technique for RMA-Analyzer [1]. We present a way to better manage the BST that is used to store all the accesses to the address space of an MPI process. We introduced a new insertion algorithm including node fragmentation and node merging. By doing so, the resulting BST should detect more errors than the reference implementation and its size is greatly reduced. For future works, we plan to enhance the static analysis proposed by Saillard et al. [14] to detect more errors at compile time. We also plan to combine this static analysis to RMA-Analyzer in order to reduce the overhead at runtime. To further help developers write MPI-RMA programs, we plan to explore code transformation at compile time. To do so, an analysis to predict where the use of MPI-RMA operations could lead to better performance compared to two-sided MPI operations may be useful. We will also explore a method to automatically transform two-sided MPI operations into one-sided operations using the LLVM compiler infrastructure [9].

## References

1. Aitkaci (T. C.), Sergent (M.), Saillard (E.), Barthou (D.) et Papauré (G.). – Dynamic Data Race Detection for MPI-RMA Programs. – In *EuroMPI 2021 - European MPI Users's Group Meeting*, Munich, Germany, septembre 2021.
2. Bachan (J.), Bonachea (D.), Hargrove (P. H.), Hofmeyr (S.), Jacquelin (M.), Kamil (A.), van Straalen (B.) et Baden (S. B.). – The upc++ pgas library for exascale computing. – *PAW17*, PAW17, New York, NY, USA, 2017. Association for Computing Machinery.
3. Chen (Z.), Dinan (J.), Tang (Z.), Balaji (P.), Zhong (H.), Wei (J.), Huang (T.) et Qin (F.). – MC-Checker : Detecting Memory Consistency Errors in MPI One-Sided Applications. – In *SC '14 : Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 499–510, 2014.
4. Diep (T.-D.), Furlinger (K.) et Thoai (N.). – MC-CChecker : A Clock-Based Approach to Detect Memory Consistency Errors in MPI One-Sided Applications. – In *Proceedings of the 25th European MPI Users' Group Meeting, EuroMPI'18*, EuroMPI'18, New York, NY, USA, 2018. Association for Computing Machinery.
5. Hilbrich (T.), Schulz (M.), de Supinski (B. R.) et Müller (M. S.). – MUST : A Scalable Approach to Runtime Error Detection in MPI Programs. – In Müller (M. S.), Resch (M. M.), Schulz (A.) et Nagel (W. E.) (édité par), *Tools for High Performance Computing 2009*, pp. 53–66, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
6. Hoeffler (T.), Dinan (J.), Thakur (R.), Barrett (B.), Balaji (P.), Gropp (W.) et Underwood (K.). – Remote Memory Access Programming in MPI-3. *ACM Trans. Parallel Comput.*, vol. 2, n2, jun 2015.

---

2. PARCOACH : <https://github.com/parcoach/parcoach>

7. Huchant (P.), Saillard (E.), Barthou (D.) et Carribault (P.). – Multi-Valued Expression Analysis for Collective Checking. – In *EuroPar*, Göttingen, Germany, août 2019.
8. Lamport (L.). – Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, vol. 21, n7, jul 1978, p. 558–565.
9. Lattner (C.) et Adve (V.). – LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation. – In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
10. Li (M.), Lu (X.), Hamidouche (K.), Zhang (J.) et Panda (D. K.). – Mizan-RMA : Accelerating Mizan Graph Processing Framework with MPI RMA. – In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pp. 42–51, 2016.
11. Li (M.), Lu (X.), Potluri (S.), Hamidouche (K.), Jose (J.), Tomko (K. A.) et Panda (D. K.). – Scalable Graph500 design with MPI-3 RMA. *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, 2014, pp. 230–238.
12. Message Passing Interface Forum. – *MPI : A Message-Passing Interface Standard Version 4.0*, juin 2021.
13. Park (M.-Y.) et Chung (S.-H.). – Detecting Race Conditions in One-Sided Communication of MPI Programs. – In *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*, pp. 867–872, 2009.
14. Saillard (E.), Sergent (M.), Aitkaci (T. C.) et Barthou (D.). – Static Local Concurrency Errors Detection in MPI-RMA Programs. – In *Correctness 2022 - Sixth International Workshop on Software Correctness for HPC Applications*, Dallas, United States, novembre 2022.
15. Schwitanski (S.), Jenke (J.), Tomski (F.), Terboven (C.) et Müller (M. S.). – On-the-Fly Data Race Detection for MPI RMA Programs with MUST. – In *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)*, pp. 27–36, 2022.
16. Serebryany (K.), Potapenko (A.), Iskhodzhanov (T.) et Vyukov (D.). – Dynamic Race Detection with LLVM Compiler. – In Khurshid (S.) et Sen (K.) (édité par), *Runtime Verification*, pp. 110–114, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

## Appendix

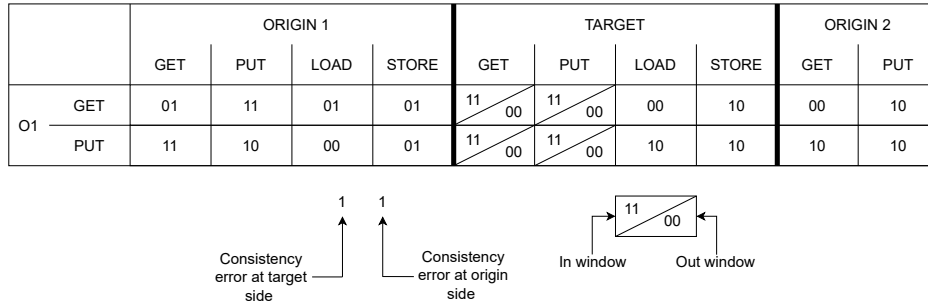


FIGURE 5 – All possible data races. Rows represent the first operation issued by Origin 1 and the columns represent another operation issued by Origin 1, Target or Origin 2. Figure 2a is described by the right bit in the cell (O1-GET, ORIGIN1-STORE) and Figure 2b is described by both bits in the upper part of the cell (O1-GET, TARGET-GET).



	Approach used	PM or OtF an.	Prof.	MPI-3	St. An.	Detect all err.	No FP	Fortran prog.
RMA-Analyzer	BST	OtF	PMPI	For load-store	Partial implem.	No error at target side	If no aliasing	
Static analysis	BFS on CFG	-	-					
MUST-RMA	MUST+ TSan	OtF	P <sup>n</sup> MPI	For load-store	Happens-before	Happens-before	Happens-before	
MC-Checker	CR from DAG	PM	Profiler					
MC-CChecker	Clock-based	PM	Profiler					
Mirror window	Mirror window	OtF	PMPI					

FIGURE 6 – Comparison of existing tools (PM : Post-Mortem, OtF : On-the-Fly, Prof. : Profiling, St. An. : Static Analysis, err. : errors, FP : False Positive, prog. : Programs, implem. : implementation).

New type \ Old type	Local_Read	Local_Write	RMA_Read	RMA_Write
Local_Read	Local_Read	Local_Write	RMA_Read	RMA_Write
Local_Write	Local_Write	Local_Write	RMA_Write	RMA_Write
RMA_Read	RMA_Read	RMA_Write	RMA_Read	RMA_Write
RMA_Write	RMA_Write	RMA_Write	RMA_Write	RMA_Write

FIGURE 7 – Resulting access type given a New\_type (in column) and an Old\_type (in row) ([1]).

---

#### Algorithm 1 Insert in BST Algorithm

---

```

1: function INSERT_BST(Acc, BST)
2:   intersection_acc ← getIntersectingAccesses(Acc)
3:   leftAccess ← Access_with_lowest_low_attribute(intersection_acc)
4:   mergeableAccesses ← getMergeableAccessesAtLeft(intersection_acc)
5:   if intersection_acc == ∅ then
6:     BST.insert(Acc)
7:     return SUCCESS
8:   if mergeableAccesses.size == intersection_acc.size then
9:     return full_merge(Acc, mergeableAccesses, BST)
10:  if mergeableAccesses.size == 0 then
11:    remain ← fragment_intersecting_Accesses(Acc, leftAccess, BST)
12:    return insert_BST(remain, BST)
13:  upperBoundAccess ← mergeableAccesses.next(intersection_acc)
14:  return insert_BST(merge_intersecting_Accesses(Acc, mergeableAccesses, upperBoundAccess, BST), BST)

```

---

---

**Algorithm 2** Fragmenting intersecting Accesses

---

**Require:**  $\text{AccB} \in \text{BST}, \text{AccA} \cap \text{AccB} \neq \emptyset$

```
1: function FRAGMENT_INTERSECTING_ACCESSES(AccA, AccB, BST)
2:    $\text{accA\_low} \leftarrow \text{low\_bound}(\text{AccA}); \text{accA\_up} \leftarrow \text{up\_bound}(\text{AccA})$ 
3:    $\text{accB\_low} \leftarrow \text{low\_bound}(\text{AccB}); \text{accB\_up} \leftarrow \text{up\_bound}(\text{AccB})$ 
4:    $\text{BST.erase}(\text{AccB})$ 
5:    $\text{intersection\_acc} \leftarrow \text{Access}(\{\max(\text{accA\_low}, \text{accB\_low}), \min(\text{accA\_up}, \text{accB\_up})\},$ 
       $\text{AccA.Type} | \text{AccB.Type},$ 
       $\text{dominantAccess}(\text{AccA}, \text{AccB}).\text{Dbg})$ 
6:    $\text{BST.insert}(\text{intersection\_acc})$ 
7:   if  $\text{accA\_low} \neq \text{accB\_low}$  then
8:      $\text{l\_acc} = \text{Access}(\{\min(\text{accA\_low}, \text{accB\_low}), \max(\text{accA\_low}, \text{accB\_low}) - 1\},$ 
       $\text{minLow}(\text{accA}, \text{accB}).\text{Type},$ 
       $\text{minLow}(\text{accA}, \text{accB}).\text{Dbg})$ 
9:      $\text{BST.insert}(\text{l\_acc})$ 
10:  return  $\text{Access}(\{\min(\text{accA\_up}, \text{accB\_up}) + 1, \max(\text{accA\_up}, \text{accB\_up})\},$ 
       $\text{maxUp}(\text{accA\_up}, \text{accB\_up}).\text{Type},$ 
       $\text{maxUp}(\text{accA\_up}, \text{accB\_up}).\text{Dbg})$ 
```

---

---

**Algorithm 3** Merging intersecting Accesses

---

**Require:** mergeable is sorted,

for element in mergeable :  $\text{element} \cap \text{Acc} \neq \emptyset$  **AND**  $\text{element.Type} == \text{Acc.Type}$ ,  
 $\text{upperBoundAccess} \cap \text{Acc} \neq \emptyset$  **AND**  $\text{upperBoundAccess.Type} == \text{Acc.Type}$

```
1: function MERGE_INTERSECTING_ACCESSES(Acc, mergeable, upperBoundAccess, BST)
2:    $\text{leftAccess} \rightarrow \text{mergeable.begin}$ 
3:    $\text{acc\_low} \leftarrow \text{low\_bound}(\text{Acc}); \text{acc\_up} \leftarrow \text{up\_bound}(\text{Acc})$ 
4:    $\text{leftAccess\_low} \leftarrow \text{low\_bound}(\text{leftAccess})$ 
5:    $\text{upperBoundAccess\_low} \leftarrow \text{low\_bound}(\text{upperBoundAccess})$ 
6:   for all element in mergeable do
       $\text{BST.erase}(\text{element})$ 
7:   if  $\text{acc\_low} > \text{leftAccess\_low}$  then
8:      $\text{leftmost\_Access} = \text{Access}(\{\text{leftAccess\_low}, \text{acc\_low} - 1\}, \text{leftAccess.Type}, \text{leftAccess.Dbg})$ 
9:      $\text{BST.insert}(\text{leftmost\_Access})$ 
10:   $\text{merged\_Access} = \text{Access}(\{\text{acc\_low}, \text{upperBoundAccess\_low}\}, \text{Acc.Type}, \text{Acc.Dbg})$ 
11:   $\text{BST.insert}(\text{merged\_Access})$ 
12:  return  $\text{Access}(\{\text{upperBoundAccess\_low}, \text{acc\_up}\}, \text{Acc.Type}, \text{Acc.Dbg})$ 
```

---

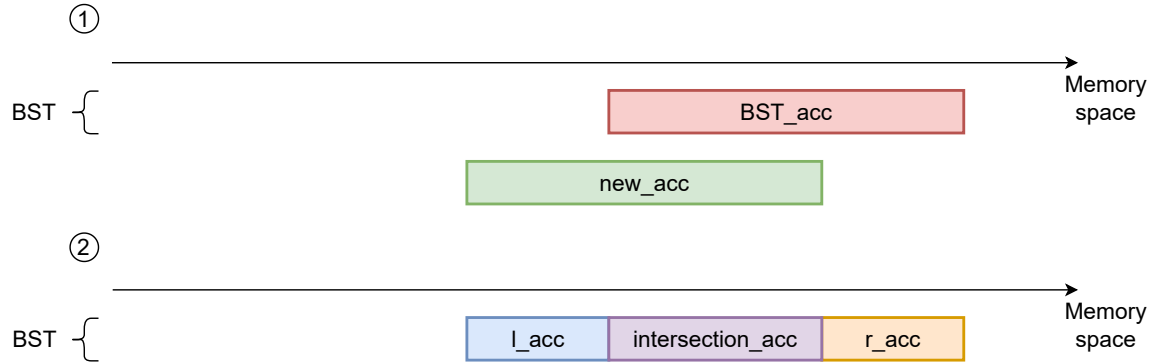


FIGURE 8 – Illustration of an execution of `fragment_intersecting_Accesses` (presented in Section 4.1). Given two intersecting accesses `BST_acc` (present in the BST) and `new_acc`, the fragmented accesses are composed of three parts : `l_acc`, `intersection_acc` and `r_acc`. These accesses are inserted in the BST and `BST_acc` is removed from the BST.

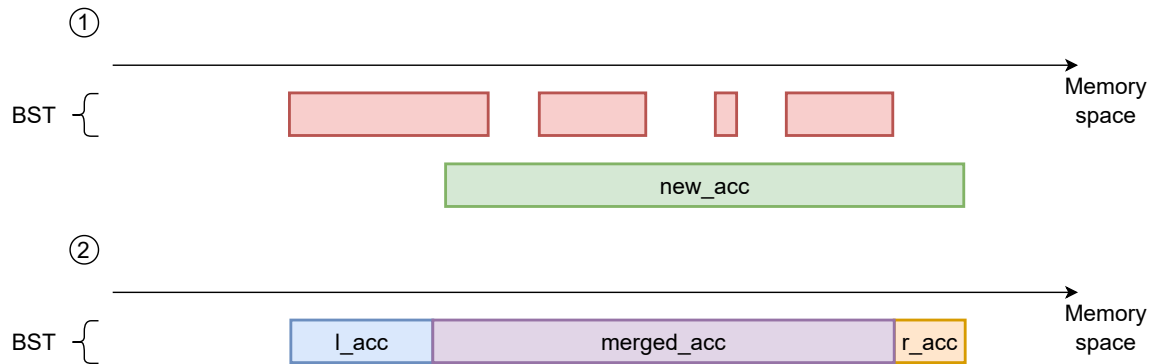


FIGURE 9 – Illustration of an execution of `merge_intersecting_accesses` (presented in Section 4.2). Given several accesses present in the BST (in red) and a new access `new_acc`, three new accesses are created : `l_acc`, `merged_acc` and `r_acc`. These accesses are inserted in the BST and all the previously present accesses are removed from the BST.

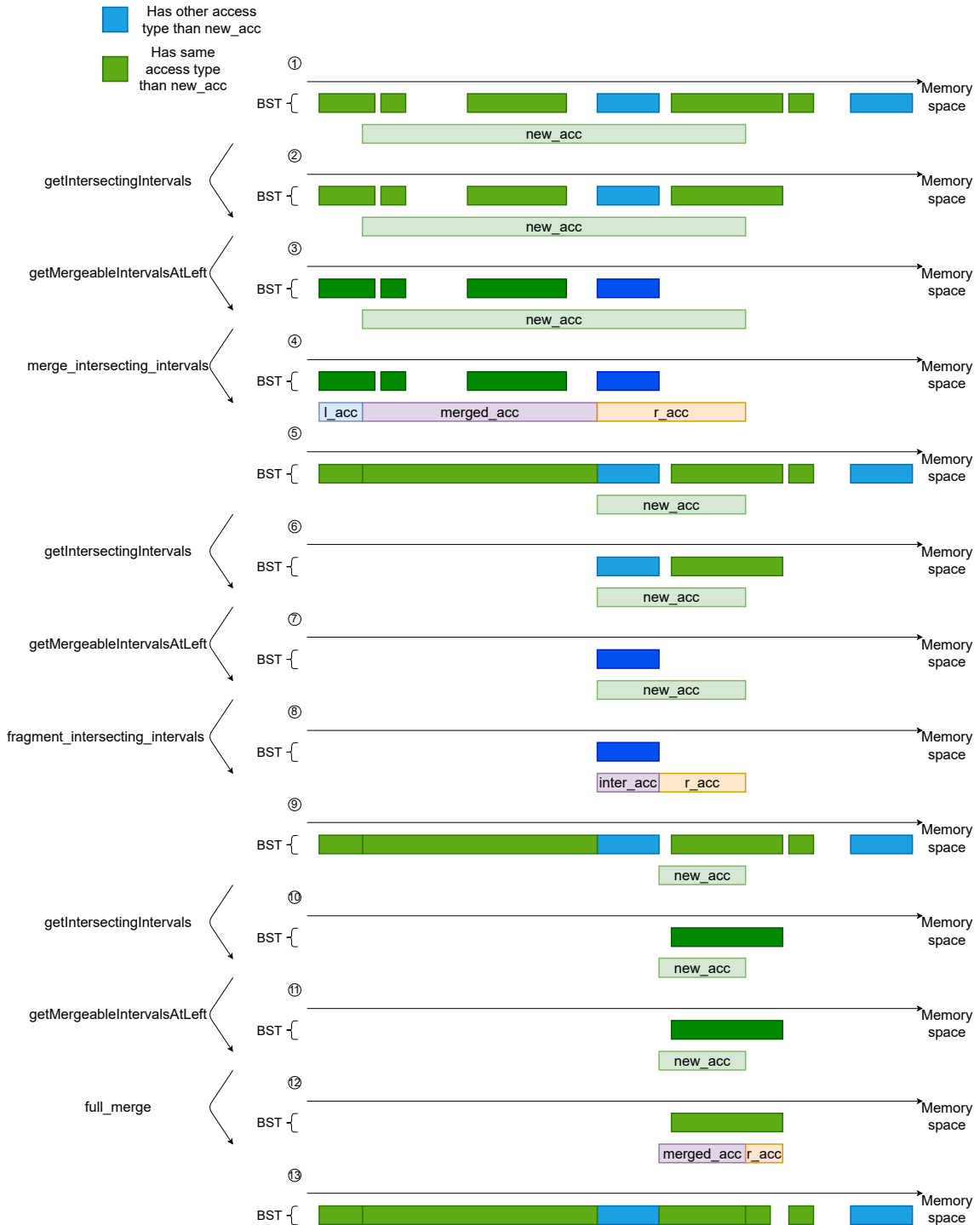


FIGURE 10 – Illustration of an execution of `insert_BST`. Given a new access `new_acc`, `new_acc` is first merged with the first accesses present in the BST that have the same access type than `new_acc` (Step 1 to Step 5). Then, when the algorithm encounters an access that has not the same access type than `new_acc`, `fragment_intersecting_intervals` is called (Step 5 to Step 9). Finally, the remaining part of `new_acc` that has not been inserted in the BST is merged with the last intersecting access in the BST (Step 9 to Step 13).