

Diese Arbeit wurde vorgelegt am  
Lehrstuhl für Hochleistungsrechnen (Informatik 12), IT Center.

# Evaluating Static Analysis Techniques to Accelerate Data Race Detection for MPI RMA

## Evaluation statischer Analysemethoden zur Performanceoptimierung von Data-Race-Erkennung für MPI RMA

Bachelorarbeit

Yussur Mustafa Oraji  
Matrikelnummer: 406408

communicated by Prof. Matthias S. Müller

Aachen, den 11. Mai 2023

Erstgutachter: Prof. Dr. rer. nat. Matthias S. Müller <sup>(1)</sup>

Zweitgutachter: apl. Prof. Dr. rer. nat. Thomas Noll <sup>(2)</sup>

Betreuer: Simon Schwitanski, M.Sc. <sup>(1)</sup>

<sup>(1)</sup> Lehrstuhl für Hochleistungsrechnen, RWTH Aachen University  
IT Center, RWTH Aachen University

<sup>(2)</sup> Lehrstuhl für Softwaremodellierung und Verifikation, RWTH Aachen University



Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Aachen, den 11. Mai 2023



# Abstract

Most high-performance computing systems utilize a distributed memory system, where a message-passing specification such as MPI is required for data communication across processes. MPI especially allows for one-sided communication, where message passing requires only one process to start the communication while the other is not required to perform a corresponding MPI call. Both standard MPI and MPI RMA are prone to data races however, requiring significant effort to find and fix. While MPI RMA data race detectors exist, they often significantly slow down program execution. This is especially the case for dynamic analysis tools which perform race detection at runtime. MUST-RMA, one such tool, can cause a slowdown of up to a factor of 16. In contrast, static tools can run cheaply at compile time with minimal overhead. The combination of both dynamic and static analysis may therefore prove useful: This thesis presents three static optimization approaches for MPI RMA data race detection based on MUST-RMA. The first approach generates a whitelist of relevant values and instructions to inspect for the dynamic tool, while others may be ignored. Though similar to the approach used in MC-Checker, the implementation is more generally applicable and extensible, for example, to additional programming languages such as Fortran. This whitelist may also be extended with additional information, more specifically on which *type* each value stored corresponds to. By checking whether or not the code only performs remote reads, writes or both additional filtering of this whitelist is possible for potential speed gain. Finally, the race detection itself may simply be delayed until the moment it is required, which is the moment the MPI RMA window is created. Additionally, the race detection may be turned off again when this window is destroyed. These optimization approaches were built on top of the LLVM framework as compile time passes, with the implementation general enough to support both C and C++ at this time. All optimizations used support interprocedural analysis, and, through the use of a modified compilation pipeline, may also be used across translation units. While introducing some false negatives, applying these optimizations provides a 2x speedup compared to normal MUST execution in most cases, with best case scenarios reaching a speedup of 4x.

**Keywords:** HPC, MPI, MPI RMA, data races



# Contents

<b>List of Figures</b>	<b>ix</b>
<b>List of Tables</b>	<b>xi</b>
<b>1. Introduction</b>	<b>1</b>
<b>2. Background</b>	<b>3</b>
2.1. MPI Remote Memory Access . . . . .	3
2.2. Related Work . . . . .	5
2.3. Detecting MPI RMA data races with MUST-RMA . . . . .	6
2.4. LLVM Analysis Framework . . . . .	8
<b>3. Optimization Approaches</b>	<b>13</b>
3.1. ThreadSanitizer Whitelist . . . . .	13
3.2. Delayed Race Detection Start . . . . .	16
3.3. Optimize using Remote Access Types . . . . .	19
<b>4. Implementation</b>	<b>23</b>
4.1. Basic ThreadSanitizer Whitelist . . . . .	23
4.2. Interprocedural and Cross-Translation-Unit Support . . . . .	27
4.3. Delayed Race Detection Start . . . . .	29
4.4. Extended Whitelist for Remote Access Type utilization . . . . .	30
<b>5. Evaluation</b>	<b>33</b>
5.1. Experiment Setup . . . . .	33
5.2. Effect on Classification Quality . . . . .	34
5.3. Overhead Study . . . . .	37
5.4. Discussion . . . . .	42
<b>6. Conclusion</b>	<b>45</b>
<b>Bibliography</b>	<b>47</b>
<b>A. Appendix</b>	<b>49</b>



# List of Figures

2.1. MUST-RMA execution overview . . . . .	8
4.1. Overview of the whitelist optimization approach implementation. . .	26
4.2. Default clang compilation pipeline . . . . .	27
4.3. Modified clang compilation pipeline for fixed whitelist support . . . .	28
4.4. Decision Tree of ThreadSanitizers instrumentation filtering using the extended whitelist. . . . .	32
5.1. Stencil Kernel performance results. Left: Absolute runtime. Right: Slowdown for each scenario. The standard deviation is used as the error. . . . .	39
5.2. Transpose Kernel performance results. . . . .	39
5.3. miniMD performance results. . . . .	40
5.4. miniVite performance results. . . . .	40
5.5. Slowdown per optimization method for each test case. All runs here without MUST. . . . .	41



# List of Tables

- 3.1. Relevant MPI RMA functions for whitelist generation. Only parameters to be marked shown . . . . . 14
- 3.2. Possibility of omitting local load instrumentation of buffer type depending on remote access type . . . . . 20



# 1. Introduction

With most HPC systems favoring distributed memory nodes, the Message Passing Interface MPI becomes the central library specification applications use to facilitate accelerated execution through high parallelization. Classically, MPI is used through a two-sided system, where data transfers between nodes require all participating ranks to call into the MPI runtime. This is the case for example for the most simple `MPI_Send / MPI_Recv` communication, but also for any collectives such as `MPI_Bcast` or `MPI_Scatter`. Often though data does not have to be processed immediately upon reception, for example when one rank finishes computation it is wasteful to wait for a receiver instead of continuing work on the larger problem. To solve this, non-blocking operations may be used which return immediately after calling; allowing the rank to continue execution while the MPI runtime handles the message transfer in the background. However, this requires extensive synchronization to make sure buffers are not reused prematurely, and still requires the involvement of both the sending and receiving ranks even though often only one party needs the communication to happen for further work.

MPI one-sided communication, or MPI Remote Memory Access (MPI RMA), solves this issue by introducing a communication model where only the communication initiator (origin) needs to call the MPI runtime, while the receiver or data source (target) completes the relevant operation passively through the MPI runtime. According to Gerstenberger, Besta, and Hoefer [4], MPI RMA additionally offers potential performance benefits as modern RDMA features of network interconnects map more easily to the RMA communication primitives. This is done by creating exposed memory buffers accessible through MPI 'window' handles. Synchronization is still needed as ranks have to be aware whether they may modify their local buffers; should a remote access be pending a conflicting local or remote access would result in undefined behavior. Thus, MPI RMA allows for the use of traditional shared memory semantics in a distributed memory environment. For example, attempting two unsynchronized writes to a remote buffer may trigger a race, or a remote read while the target is writing data to the exposed buffer locally. In general, any two operations with one writing access and one remote access can trigger a data race. This can be the case for a remote read and a local write, but also for a remote write and local read or two remote access where at least one is writing. Worsening the issue is the opaque nature of data races; as most will not cause an application crash but instead falsify results in a subtle manner, only making their presence known during evaluation.

## 1. Introduction

Due to the difficulty of providing correct synchronization for every control flow of each rank, many tools have been developed to assist in debugging these issues. Examples include MC-Checker [2] and MUST-RMA [16], which try to detect data races during runtime. They are therefore known as 'dynamic analysis' tools. In contrast, static analysis tools such as the tool developed by Saillard et al. [15] attempt to detect data races prior to runtime directly on the source code (or a transformed version thereof, mostly compile-time intermediate representations). Utilizing dynamic tools allows for a high accuracy analysis at the cost of significant runtime overhead, while static tools are characterized by low accuracy but negligible compile-time overhead<sup>1</sup>. During debugging rapid prototyping is required, which is inherently incompatible with the high overhead introduced by dynamic analysis tools. On the other hand, the low accuracy of static analysis tools makes their use redundant; there is no need to use an analysis if it does not supply relevant information.

Using static analysis differently may solve this issue: Instead of performing the race detection statically, it may be prudent to instead assist an existing dynamic tool through the information gained during compile-time. Static analysis tools have the advantage of being able to look ahead and modify code before execution, and may do so cheaply at compile time. Their weakness lies in the inability to know *which* data is stored where, and thus which execution path is chosen during runtime as a result. Improving accuracy is not fruitful due to the already great accuracy many dynamic tools possess. Instead, this thesis explores different static optimization approaches in order to accelerate the dynamic race analysis of MPI RMA.

We present three optimization methods on top of the MUST-RMA dynamic data race detector, built on top of the LLVM framework. These aim to provide a speedup compared to normal execution with race detection enabled, while minimizing potential detection accuracy losses. Additionally, some of these optimization approaches also serve as a basis for additional optimizations in the future, presented throughout the course of this thesis. Their implementation is compatible with code written in C and C++, and may also be extended to any language supported in the LLVM Intermediate Representation such as Fortran.

We will first establish required background information in Chapter 2, after which some possible optimizations are discussed in Chapter 3. Chapter 4 will dive into the practical aspect of our optimizations, and the feasibility of their implementation. In Chapter 5 the effect on classification quality as well as overhead are evaluated. Finally, Chapter 6 summarizes our findings and provides avenues for further work.

---

<sup>1</sup>Section 2.3 provides detailed measurements on two examples

## 2. Background

This chapter introduces required background knowledge regarding MPI RMA as well as the tooling used for the analysis. Section 2.1 briefly introduces programming with MPI RMA as well as data races and their semantics. The automatic detection of these issues is discussed in Section 2.3. Additionally, the operation of the tool chosen as the analysis implementation basis is described. Finally, Section 2.4 deals with the analysis framework and the transformed source our analysis will use.

### 2.1. MPI Remote Memory Access

In contrast to the more commonly used two-sided MPI communication, where both the sender and receiver actively participate through the use of MPI calls such as `MPI_Send` / `MPI_Recv` or collectives such as `MPI_Bcast`, `MPI_Scatter`, MPI Remote Memory Access (MPI RMA) is a one-sided alternative. When using MPI RMA, only one of the sender or receiver actively initiates the communication (the *origin*), and the other (the *target*) completes it passively through the MPI runtime. MPI RMA communication happens through exposed memory buffers called MPI 'windows', created by `MPI_Win_create` or `MPI_Win_allocate`. For `MPI_Win_create`, an existing memory buffer is passed to MPI to make available for other ranks, while `MPI_Win_allocate` creates a new buffer and allocates the required memory automatically. The creation call must be used by all ranks involved with this window.

A rank may access another ranks' window by using `MPI_Get`, which reads from the exposed buffer, `MPI_Put`, which writes to the exposed buffer, or `MPI_Accumulate`, which is an atomic<sup>1</sup> remote update operation. All MPI RMA memory accesses must happen within an 'epoch', where an epoch denotes a timeframe during which RMA operations are used. Most importantly for the understanding of this thesis is that all RMA window access calls are non-blocking. Therefore, the memory access does not finish with the call returning; instead a synchronization mechanism is needed to ensure the completion. Completing an epoch implicitly completes any pending operations on an RMA window. This may happen automatically when creating a new epoch, or must be done explicitly depending on the synchronization mechanism used. For example, the MPI standard guarantees completion of all pending remote

---

<sup>1</sup>On supported datatypes.

## 2. Background

memory accesses when a new epoch<sup>2</sup> is created using `MPI_Win_fence`, which is a collective call for all ranks involved with the window. Simple Mutex-like synchronization is possible through the use of `MPI_Win_lock` and `MPI_Win_unlock`. Here `MPI_Win_lock` creates an RMA access epoch, and calling `MPI_Win_unlock` ensures completion. Different to standard mutexes however `MPI_Win_lock` is only blocking if the calling rank is also the target rank. `MPI_Win_unlock` always blocks until target completion. Additionally, `MPI_Win_lock` can be both configured to be shared (meaning other ranks may also acquire the lock concurrently) or exclusive (calling rank gets exclusive access, or is blocked until all other ranks release the lock). Finally, Post-Start-Complete-Wait (PSCW) allows programmers to distinguish and specify access and exposure epochs. Using `MPI_Win_start`, *select* ranks start an access epoch and are allowed to use RMA communication calls. They may only access exposed memory, which must be explicitly marked by a corresponding `MPI_Win_post` call, creating an exposure epoch. When all accesses are complete, `MPI_Win_complete` closes the access epoch. Conversely, `MPI_Win_wait` closes the exposure epoch. The synchronization methods using `MPI_Win_fence` and PSCW are called *active synchronization* as they require target process involvement. `MPI_Win_fence` is a collective call, and PSCW requires the target to start and complete the exposure epoch. Only using the Lock/Unlock mechanism *passive synchronization* is achieved, where target process involvement is not required at all. `MPI_Win_lock` creates an access epoch on the target process, and only has to be called by the origin. The concept of an exposure epoch is not utilized when using `MPI_Win_{lock,unlock}`. Thus, the target is not needed to open the epoch, and communication may begin without active involvement from the target's side.

Due to the complexity of correct synchronization their use has become quite error-prone, and missing one or incorrect usage may lead to a data race. An MPI RMA data race happens between two concurrent accesses to exposed memory, either using two unsynchronized remote accesses (such as two sequential `MPI_Put` operations on the same memory) or one local and one remote accesses (such as an `MPI_Get` on the origin and a local write on the target). A data race can also occur solely on the origin side, for example, by modifying a buffer used by a pending remote access operation. In general, a local buffer race is defined as a race where both the RMA operation as well as the conflicting memory access (whether from an additional RMA operation or local access) happen on the same rank. Conversely, any race where this property does not hold is defined as a remote race. It also means that only the origin may experience a local buffer race. Consider Listing 2.1, which is an example code with two ranks. A window for a single integer is allocated, after which `MPI_Win_fence` is used to create an epoch. Rank 0 writes a value from a local buffer to the exposed window, and rank 1 prints the exposed buffer value. As these two accesses have no synchronization between them and can happen concurrently, and as one of them is a writing access, this code contains a data race. Note that the two accesses on the `test` local buffer using the `MPI_Put` and following `printf`

---

<sup>2</sup>`MPI_Win_fence` automatically starts both a new access and exposure epoch

Listing 2.1: MPI RMA data race  
(Simplified C)

---

```

1 MPI_Comm_rank(&rank);
2 int* buf;
3 int* test =
4     malloc(sizeof(int)*2);
5 MPI_Win win;
6 MPI_Win_allocate(&buf, &win);
7 MPI_Win_fence(win);
8 if (rank == 0) {
9     test[0] = 42;
10    test[1] = 10;
11    MPI_Put(&test[0], win);
12    printf(test[0]);
13 } else {
14    printf(*buf);
15 }
16 MPI_Win_fence(win);

```

---

Listing 2.2: Listing 2.1, fixed

---

```

1 MPI_Comm_rank(&rank);
2 int* buf;
3 int* test =
4     malloc(sizeof(int)*2);
5 MPI_Win win;
6 MPI_Win_allocate(&buf, &win);
7 MPI_Win_fence(win);
8 if (rank == 0) {
9     test[0] = 42;
10    test[1] = 10;
11    MPI_Put(&test[0], win);
12    printf(test[0]);
13    MPI_Win_fence(win);
14 } else {
15    MPI_Win_fence(win);
16    printf(*buf); // 42
17 }

```

---

do not cause a data race, as both are reading accesses. Listing 2.2 provides a fix by ensuring the completion of writing access of rank 0 before the reading access of rank 1 by creating a new epoch using `MPI_Win_fence`. In essence, we are moving the `MPI_Win_fence` call after the branch into the branch itself. Finding data races can be hard as they do not necessarily lead to a program crash, but instead more often lead to falsified results; with these occurring nondeterministic. Thus, data race detection tools become important for MPI RMA development.

## 2.2. Related Work

Only minimal work has gone into the topic of optimizing RMA race detection using static analysis. The closest to our approach is MC-Checker [2], which utilizes a subcomponent 'ST-Analyzer' to evaluate relevant variables for race detection. This is done using Clang, a compiler frontend for LLVM. While similar to the approach presented in Section 3.1, apart from MC-Checker and ST-Analyzer not being available publicly, the implementation differs significantly as do the possible use cases. Most importantly our analysis is built as a compile time pass for LLVM, allowing for easier extensibility. For example, the analysis is not limited to the C programming language, but may also run on C++ and any other language that can be reduced to the LLVM Intermediate Representation (LLVM IR). Fortran uses different calling semantics when reduced to LLVM IR as well as differing names for the MPI functions. This causes our analysis to fail, but extending support is trivial, though out of scope for the proof of concept presented in this thesis. Additionally, the pass

## 2. Background

presented here may be used by any tool interfacing with ThreadSanitizer, and modified for different distributed memory paradigms. Finally, this thesis also presents 2 other optimization approaches not related to that used by MC-Checker.

While the application differs, there are other tools combining static and dynamic analysis as well. PARCOACH [14] is one tool combining these analysis types in order to detect deadlocks with collective calls in MPI. However, not only does PARCOACH not support MPI RMA communication, it also uses the static analysis for error detection primarily and not for optimization of an existing dynamic tool. Instead, the static phase of PARCOACH execution already performs deadlock detection, but also instruments the code for MPI collective calls. Then during runtime, errors are thrown should a deadlock occur at one of these collective calls.

There is also the tool developed by Saillard et al. [15] detecting local data races in MPI RMA code. This is purely static though, and quite limited due to the focus on local data races. Additionally, it does not support interprocedural functionality as the implementation in this thesis does.

### 2.3. Detecting MPI RMA data races with MUST-RMA

As working with highly parallel code is hard due to the difficulty of keeping track of the numerous different execution paths for each rank, many tools have been written to assist in race detection. They can be roughly categorized into static, dynamic or hybrid analyzers. Static analyzers run either directly on the source code or on a transformed version of it, and run race detection prior to execution. For example the tool by Saillard et al. [15] detects local buffer races with RMA fully statically. In contrast, dynamic analysis tools must be used during program runtime. Examples include MUST-RMA [16], which we will be using through the course of this thesis, and the tool developed by Aitkaci et al. [1]. Hybrid analyzers use both statically collected information as well as data gathered during runtime to perform the race analysis. Chen et al. [2] use static analysis for preprocessing possible RMA memory locations prior to runtime to pass relevant memory locations to the runtime, allowing the runtime to discard the rest. The actual race analysis then only happens during runtime. Static analysis tools tend to be significantly less accurate than dynamic tools, as there is more information available during runtime. This comes at a cost however, as dynamic tools are quite slow compared to static tools. The MPI Bugs Initiative [9] is a test suite and accuracy benchmark for MPI error detection (notably not limited to data races). Their results [8] indicate this relationship between static and dynamic tools well, with the dynamic tool MUST having an accuracy rating of 96%, while static tools such as PARCOACH [14] only 38%<sup>3</sup>. PARCOACH has an

---

<sup>3</sup>PARCOACH is a hybrid tool, but can be run statically. This was done in this instance.

$\approx 10\%$  overhead on compile time [14], while MUST can have a runtime overhead of up to 16x [6]. We have selected MUST as our implementation basis, and our goal is therefore to accelerate MUST-RMA through our static analysis.

It is therefore important to understand its operating principle. MUST is a general purpose error detection tool for MPI, supporting many different issues such as deadlocks, data type mismatches and leak checks [7]. For our usecase however, only the data race detection is needed. In its default configuration, only data races for two-sided communication are detected. Instead, we must use the currently in-development MUST-RMA extension to MUST, developed by Schwitanski et al. [16]<sup>4</sup>. MUST-RMA detects data races in one-sided MPI code by generating 'concurrent regions' using the consistency relation  $\xrightarrow{co}$ , which denotes completion semantics of RMA operations, and the happens-before relation  $\xrightarrow{hb}$ , which denotes synchronization between ranks [16]. These two definitions are combined to the 'consistent happened-before relation'  $\xrightarrow{cohb} = \xrightarrow{hb} \wedge \xrightarrow{co}$ . Finally, a data race is defined to be any two conflicting<sup>5</sup> operations  $a, b$  where  $a \not\xrightarrow{cohb} b \wedge b \not\xrightarrow{cohb} a$ , which can be simply read as two operations where one is not guaranteed to finish before the other, and no synchronization mechanism forbids their concurrency.

Now that we have established the theory of race detection, we move on to the implementation of MUST-RMA. To establish the  $\xrightarrow{cohb}$  relation, MUST-RMA hooks into every RMA memory access and synchronization call. Then, MUST-RMA hands over this information to ThreadSanitizer [17], a data race detector for pthreads and C++11 threads. ThreadSanitizer consists of two parts: The ThreadSanitizer runtime and the compile-time pass. The pass simply instruments a call to the ThreadSanitizer runtime for every memory access, allowing it to keep track of all local memory accesses. Then, the ThreadSanitizer runtime keeps track of the current execution context using shadow memory, storing information such as synchronization between memory accesses, remote availability and pending operations. Using this information, ThreadSanitizer is capable of running race detection on multithreaded applications. ThreadSanitizer also contains an annotation framework, allowing other tools to interface with ThreadSanitizer. MUST-RMA uses these annotations to take over the handling of synchronization mechanisms and remote accesses for ThreadSanitizer, while allowing ThreadSanitizer to perform the race checks, effectively extending ThreadSanitizers use to MPI RMA. Now, during execution, ThreadSanitizer has all required information available; with the RMA operations provided by MUST and local memory information from the instrumented calls to the runtime. Finally, race detection runs for each jump into the runtime, whether from MUST or from instrumented code. If the new operation is conflicting with another currently pending operation in shadow memory, a data race is reported. Otherwise, the runtime returns to normal execution. Figure 2.1 provides a rough overview of the entire

<sup>4</sup>Source available and given in the cited paper

<sup>5</sup>At least one of the operations must be writing, and at least one must be associated with an MPI RMA call.

## 2. Background

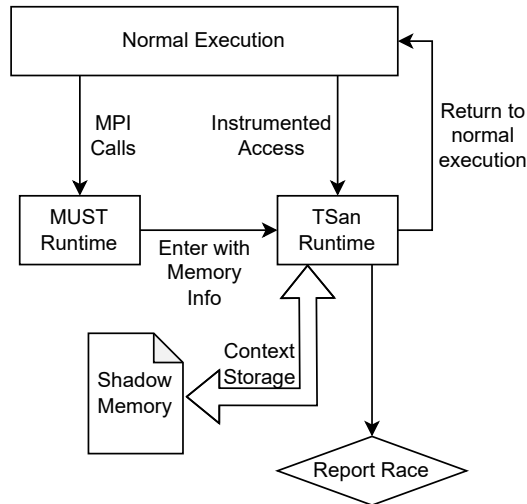


Figure 2.1.: MUST-RMA execution overview

runtime stack.

By reviewing the detection setup it becomes apparent that the ThreadSanitizer instrumentation is most costly, and may be responsible for most of the slowdown introduced. This sentiment is echoed by the results of [16]. This is due to the instrumentation of *every* memory access, no matter if relevant or not. As shadow memory is created for each memory access as well, memory usage is also increased significantly. In total, Serebryany and Iskhodzhanov [17] measured anything from a 20x to 50x slowdown<sup>6</sup>, and increased memory consumption by a factor of around 3x to 4x. As such, it is our goal to minimize the amount of instrumented memory accesses or otherwise make sure that these runtime jumps have less of an effect on performance.

## 2.4. LLVM Analysis Framework

In order to perform the static optimization analysis, our implementation will be leveraging the LLVM analysis framework as a compile time pass. The LLVM framework has been used by many static analysis tools such as the tool by Saillard et al. [15] presented in 2.3, and is easily extensible, thus solidifying our choice. We also evaluated writing the analysis on top of Clang, allowing us to work nearer to source code level. However, this would have proved challenging: By being nearer to source code, we are also exposed to more of the language specific constructs. Modelling all of them would be difficult and error-prone. Additionally, as ThreadSanitizer also uses the LLVM framework to implement the compile time instrumentation, we can

<sup>6</sup>Data from an old version of ThreadSanitizer. Current performance is largely undocumented but found to be around 2x-5x empirically on our test cases. Real-World code slowdown may skew higher.

directly influence ThreadSanitizer without building an intermediate layer. Finally, building an LLVM pass allows us to work on the LLVM intermediate representation (LLVM IR), to which *all* languages supported are reduced to. This allows us to not only support C code, but also C++, and any other language that may be reduced to LLVM IR. Exceptions include languages using different calling heuristics in the IR such as Fortran. Implementation is not difficult, but deemed out of scope for this thesis. The static analysis tool by Saillard et al. [15] is an example of this, supporting both Fortran and C by leveraging the advantages of the LLVM framework.

With an LLVM pass, we can access all information in the LLVM IR as well as modify it. The LLVM IR uses a single static assignment form. Each instruction returns a single 'LLVM Value', and the value may not be overwritten at any point. While simplifying the syntax, this also introduces a lot of aliasing between values, as any modification to a value generates a new value and cannot replace the existing one. Every instruction is written by itself; to use the return value of another instruction it needs to be in a separate line. This eases modification as we can simply replace the operation that generated the parameter we want to replace, if needed. Because of this single static assignment form there exists a one to one correspondence of a value and the generating instruction. Functions and global variables may carry attributes. Important for us, ThreadSanitizer utilizes the `sanitize_thread` attribute to know which functions to instrument. Additionally, the `no_sanitize_thread` attribute can be used to forbid ThreadSanitizer from instrumenting specific global variables or function contents.

When compiling, the source files are transformed to IR and then compiled to object files. This also means that for each source file one IR is generated, as we have one object file per source file as well. An LLVM pass may operate for example on loops, functions or on a complete IR, which is the most general case - but not *multiple* IR files at once, which is an important limitation. Strictly speaking, LLVM differentiates between passes and analyses, where a pass may modify the IR, and an analysis can report back information as a result. More specifically, the results produced by an LLVM analysis are to be used by different LLVM passes or analyses, though generally not for the end user. The LLVM passes are the more commonly used and known compile passes such as all optimization passes, instrumentation passes (including ThreadSanitizer) or function inliners. For simplicity, we use these terms interchangeably, as they offer the same functionality otherwise. To run an LLVM pass it has to be added to the pass manager, after which a flag has to be added to Clang<sup>7</sup> (the compiler frontend supplied by LLVM). It can then be invoked by using the flag while compiling. Options may be added to specific passes as well; options may be passed to the LLVM backend by using the `-mllvm` flag followed by the option on the Clang frontend.

Listing 2.3 shows an excerpt of the corresponding LLVM IR, the code inside the branch for `rank == 0`, including ThreadSanitizer instrumentation. One line is one

---

<sup>7</sup>To compile Fortran, the Flang project is used. Clang supports C and C++

## 2. Background

Listing 2.3: LLVM IR excerpt of Listing 2.1, rank == 0 branch

---

```
1 ; Corresponding C code:
2 ; ---
3 ; test[0] = 42;
4 ; test[1] = 24;
5 ; MPI_Put(&test[0], 1, MPI_INT, 1, 0, 1, MPI_INT, win);
6 ; printf("Test: %d", test[0]);
7 ; ---
8 ; %9 is 'test' buffer pointer
9 ; %10 is RMA window handle
10 %20 = load ptr, ptr %9, align 8
11 %21 = getelementptr inbounds i32, ptr %20, i64 0
12 call void @__tsan_write4(ptr %21)
13 store i32 42, ptr %21, align 4
14 %22 = load ptr, ptr %9, align 8
15 %23 = getelementptr inbounds i32, ptr %22, i64 1
16 call void @__tsan_write4(ptr %23)
17 store i32 24, ptr %23, align 4
18 %24 = load ptr, ptr %9, align 8
19 %25 = getelementptr inbounds i32, ptr %24, i64 0
20 call void @__tsan_read8(ptr %10)
21 %26 = load ptr, ptr %10, align 8
22 %27 = call i32 @MPI_Put(ptr noundef %25, ...)
23 %28 = load ptr, ptr %9, align 8
24 %29 = getelementptr inbounds i32, ptr %28, i64 0
25 call void @__tsan_read4(ptr %29)
26 %30 = load i32, ptr %29, align 4
27 %31 = call i32 (ptr, ...) @printf(..., i32 noundef %30)
```

---

instruction, with the return value on the left side of the equality. Should there not be an equality sign, the instruction returns `void`<sup>8</sup>. Comments begin with a semicolon. The aliasing issue becomes much more apparent in the LLVM IR: consider values `%20`, `%22`, `%24` and `%28`. Each of these have the same defining instruction, including the same parameters. As mentioned, the LLVM IR uses the single static assignment form, and thus once a value is created it can no longer be altered. We conclude that all three of these values are aliased, as none of their parameters changed, and the instruction does not have side effects due to it being a simple load. The ThreadSanitizer instrumentation is also visible now. Values `%21` and `%23` each represent the pointer to the memory address of the entries in the local `test` buffer. Just before their values are written two lines afterward, an instrumented ThreadSanitizer call is present (`__tsan_write4`)<sup>9</sup>. During runtime, this informs ThreadSanitizer about the memory modification. It also becomes clear that these calls are often unneeded. Consider value `%23` and `%29`. As mentioned, `%23` is one of the local writes also present in the C code, namely the second buffer entry. But due to the second buffer entry

---

<sup>8</sup>`void` is treated as a normal return value. This way, algorithms do not have to differentiate.

<sup>9</sup>The 4 at the end just indicates that 4 bytes are written. This is system dependant.

never being used to send or receive, and is not exposed, there is no need for its instrumentation. %29 is aliased to %21 and %25, the first local buffer value, as all parameters used are also aliased and the same instruction is used. This means that %29 is actually used for RMA communication, as its aliased value is used in the `MPI_Put` call. However, the instrumented `__tsan_read4` call just after the definition of %29 is still unneeded. Reading from a buffer used for `MPI_Put` does not cause a data race due to neither being a writing access. %10 and %26 are the easiest to argue should not be instrumented. As they are only the window handles, no data stored or loaded can lead to a data race. The following chapter will describe multiple optimization approaches which aim to minimize these unneeded calls and thus improve performance.



## 3. Optimization Approaches

This thesis introduces three different static approaches to optimize the dynamic race analysis. While the implementation focuses on the practical application with MUST-RMA, the approaches presented here are theoretically applicable to any race detector requiring only a subset of the local buffer information. The first will be introduced in Section 3.1, and, similarly to [2], marks relevant memory accesses to generate a whitelist. Another approach is to delay the race detection until required, which is explored in Section 3.2. Finally, if the source only uses remote writes or remote reads exclusively, additional optimization may be possible. Section 3.3 describes an optimization possible only on code exhibiting these specific access patterns.

### 3.1. ThreadSanitizer Whitelist

As ThreadSanitizer's instrumentation is generated for each memory access, it is exceptionally expensive during runtime (see Section 2.3). However, not every local memory access is relevant for MPI RMA race detection. Many memory accesses such as temporary variable assignments and reads, index values in loops or configuration parameters most likely never interact with memory regions relevant to MPI RMA. In general, the only memory regions relevant to MPI RMA are those exposed through a window creation call, or local buffers used for remote reads or writes. During runtime, differentiating between these memory accesses requires keeping track of the memory state using shadow memory (see Section 2.3), which is expensive in both time and memory. Buffers are created at one point and may be used for remote accesses at any time. The runtime thus has to be informed immediately when any memory is accessed, as at any future point it may be used for MPI RMA remote operations.

This limitation does not exist for static analysis. When using static analysis the entire source is available at once, and it is therefore possible to decide whether a memory location has any possibility of interacting with exposed memory. ThreadSanitizer currently instruments every memory access on the possibility of interacting with exposed memory. Instead, the static analysis could look at all MPI RMA operations used, and check for relevant memory locations from here on out. More specifically, the optimization suggested here is to check for any MPI RMA operation that interacts with exposed memory (whether creating or accessing it), and to only mark the memory regions used there for instrumentation by ThreadSanitizer.

### 3. Optimization Approaches

Window Creation
MPI_Win_create(void *base, ...)
MPI_Win_allocate(..., void *baseptr, ...)
Remote access write
MPI_Put(const void *origin_addr, ...)
MPI_Rput(const void *origin_addr, ...)
Remote access read
MPI_Get(const void *origin_addr, ...)
MPI_Rget(const void *origin_addr, ...)
Mixed access
MPI_Accumulate(const void *origin_addr, ...)
MPI_Raccumulate(const void *origin_addr, ...)
MPI_Get_accumulate(const void *origin_addr, ..., void *result_addr, ...)

Table 3.1.: Relevant MPI RMA functions for whitelist generation. Only parameters to be marked shown

This alone would not be enough to detect data races. Memory pointers may be aliased at will in many programming languages such as the ones we attempt to support, C and C++. To solve this, we use a recursive approach. When an MPI RMA call creating or accessing exposed memory is encountered during analysis, we first mark the relevant memory pointer. For the example in Listing 2.1, the two MPI RMA calls inspected are `MPI_Win_allocate(&buf, win)` and `MPI_Put(&test[0], win)`. Both `&buf` and `&test[0]` are therefore marked. `win` is just the window handle, an identifier used to differentiate between windows, and therefore does not need to be marked. The other MPI RMA calls are synchronization calls and, while relevant to race detection, do not represent an MPI RMA operation on memory. A full list of relevant functions and marked parameters is given in Table 3.1 Then all 'users', which we define as values generated using a value (the 'use'), of the memory pointer are marked as well. Recall from Section 2.4 that the LLVM IR the analysis is operating on uses a single static assignment form. Thus, any instruction or modification on a memory pointer will be able to be checked one by one, and they will each generate another value. We then mark these generated values as well. These steps are performed recursively on the new generated values, and will in the end result in a finalized list of relevant memory accesses. Listing 3.1 shows a pseudocode overview of the presented algorithm.

Any memory access not in this list is guaranteed to never interact with exposed memory, and may never be used as a local buffer for remote accesses. We make the following claim: Any memory access not in this list cannot lead to a data race. Therefore, it is a functional whitelist for instrumentation by ThreadSanitizer, where any memory access not present does not have to be instrumented. Assume a value  $v$  is used with an exposed buffer  $e$  in an instruction causing a data race. Our analysis marks memory pointers used in relevant MPI RMA accesses, and therefore

Listing 3.1: Basic Whitelist generation pseudocode

---

```

1 main() -> void:
2 List whitelist = {}
3 for Instruction I in code:
4     if I is call to any {MPI_Get, MPI_Put, MPI_Rget, MPI_Rput,
5                         MPI_Accumulate, MPI_Raccumulate,
6                         MPI_Get_accumulate,
7                         MPI_Win_create, MPI_Win_allocate}:
8         Value v = getRelevantParameter(I)
9         whitelist = whitelist  $\cup$  recurse(v)
10 return whitelist

```

---

Listing 3.2: Recursive step of whitelist generation

---

```

1 recurse(Value v, Type t) -> List<Value>:
2 List temp = {v}
3 for Use u of v:
4     if u is function call and getFunc(u) is defined in code:
5         temp = temp  $\cup$  recurse(getParam(u))
6     temp = temp  $\cup$  recurse(getUser(u))
7 return temp

```

---

the generated list  $W$  includes at least the value  $e$  already. For any transformed version  $v'$  of  $v$  and  $e'$  of  $e$ , there must be an instruction using  $v$  or  $e$  respectively, which returns the modified version. During the recursive step, we mark every user of the exposed memory / local buffer. Thus, during this step we will encounter the relevant instruction and  $W = W \cup e'$  at some point. Finally, at some point the actual interaction instruction occurs, where an instruction  $c$  will use  $v'$  and  $e'$  together. This may or may not trigger a race at runtime, but: Which access it is, and what kind of instruction is irrelevant to this approach; we are only interested in the instrumentation. The race reports will be generated by ThreadSanitizer should a remote access be pending, and this instruction conflicts with it; and the access will be ignored if not. As we do not have information on pending operations during static analysis, we must only make sure that in these kinds of instructions, the instrumentation is present. Again, as we are using LLVM IR, this instruction also generates a value. Thus, this value  $c$  is a user of  $e'$ , and will be marked during the recursive step. With this, we have marked the relevant memory access. As we have kept these steps general, but linear, our analysis holds for any strictly intraprocedural code. The analysis is easily extended to interprocedural support: Should a user encountered during the recursive step be a function call, and this function call is defined in source, the relevant parameter at the function *definition* is marked as well. Notice how  $v'$  and  $v$  were however not marked. This is intentional: We are uninterested in these values, as the instructions generating them do not represent a data race, there is no need to instrument them. Only the 'possible

### 3. Optimization Approaches

Listing 3.3: Whitelist applied to Listing 2.1. Relevant RMA functions are bold, whitelist entries are underlined. Every line not marked in any way is not instrumented

---

```
1 MPI_Comm_rank(&rank);
2 int* buf;
3 int* test = malloc(sizeof(int)*2);
4 MPI_Win win;
5 MPI_Win_allocate(&buf, &win);
6 MPI_Win_fence(win);
7 if (rank == 0) {
8     test[0] = 42;
9     test[1] = 10;
10    MPI_Put(&test[0], win);
11    printf(test[0]);
12 } else {
13    printf(*buf);
14 }
15 MPI_Win_fence(win);
```

---

conflict instruction'  $c$  must be marked. We also marked  $e$  and  $e'$ , but these do not necessarily have the potential to cause data races. In the IR, there are not only load and store instructions, but many possibly benign instructions that transform the LLVM value, but do not interact with memory. The generated whitelist is therefore still an overapproximation.

Listing 3.3 shows what the whitelist applied to 2.1 would look like. Note that the analysis actually runs on the LLVM IR. This is therefore just a simplified view. The algorithm would first mark the values in the IR corresponding to the local `test` and exposed `buf` buffers due to their use in some of the RMA functions listed in Table 3.1. Then, each of their uses would be marked recursively. In the IR excerpt (Listing 2.3) this would mark almost every value, as almost all somehow interact with the exposed buffer. Even the second buffer entry value would be marked, as it originates from the same pointer. However, it does not mark every value. This optimization removes the need for the instrumentation of values `%10` and `%26` (the window handle) of Listing 2.3. While not visible in the excerpt, it also removes the need of instrumentation at the beginning of the code during initialization, such as setting the `rank` variable and reading it in the branch, or the instrumentation of the `argc` and `argv` variables.

## 3.2. Delayed Race Detection Start

Statically detecting exactly only the memory accesses relevant to MPI RMA is near impossible. The approach presented in Section 3.1 is still an overapproximation.

Instead, we attempt a different optimization. When running tasks on a cluster, it is very likely to deal with data of significant size. This data must also often first be read into memory and/or otherwise preprocessed. During this time, ThreadSanitizer will also cause significant slowdown; especially if huge amounts of data is read into memory as ThreadSanitizer will instrument each memory access here. While this will most likely be done in a loop (reading from file to buffer, clearing buffer, repeat), and thus there will be very few instrumented calls present, they will be called very frequently. Additionally, since these buffers will likely be worked on during the compute section of the program, the whitelist approach will likely mark the preprocessing step and therefore not improve the preprocessing slowdown.

MPI RMA has to be initialized as well. First, `MPI_Init` has to be called to be able to use any MPI operation at all. Then, there must be at least one window for MPI RMA communication to occur. While the `MPI_Init` call will likely happen close to program initialization, the MPI RMA window creation call is more likely to be close to the compute kernel as this is where the MPI communication first becomes relevant. One important realization is this: There can not be a data race if there is no exposed window buffer. The naive approach is therefore to filter out instrumented calls prior to window creation. This is not possible, as in static analysis such a before-after separation is unknown. Consider the case where the MPI window is created in a separate function, which is called multiple times for a program requiring multiple windows. A data race could occur early in this function only in subsequent iterations, which cannot be detected statically. In general, the timing relation of two instructions can only be determined statically in trivial cases, and so this approach does not work. Additionally, the existence of an RMA window is unknown statically, only the instructions generating them are known.

The goal of this thesis is to optimize the runtime of dynamic tools statically; this also means the analysis may interact with the runtime counterpart. We already know that the instrumentation is the most expensive part. But it is only a function call, the expensive part happens *during* the function call, not the call itself. And so, we amend the approach: Instead of trying to remove instrumented accesses prior to window creation, amend the runtime to exit early if no window exists at the time. ThreadSanitizer has no knowledge of MPI RMA windows, but it does not need to. As part of this thesis, the ThreadSanitizer runtime is modified for the race detection to be disabled by default, and the annotation interface is extended with a function to start the race detection for all processes. Then, using static analysis this annotation is instrumented to each window creation call. The call is cheap as it only sets a global flag in the ThreadSanitizer runtime, and only once; it does nothing on subsequent calls. This works as the window creation calls are collective; if not one rank may run ahead and create the window, causing slowdown on every other rank. As they are though, a single, shared flag across the ranks is enough to perform this optimization.

Now, should an expensive, instrumented preprocessing step occur prior to the com-

### 3. Optimization Approaches

Listing 3.4: Whitelist and delayed race detection applied to Listing 2.1. Added call is highlighted.

---

```
1 MPI_Comm_rank(&rank);
2 int* buf;
3 int* test = malloc(sizeof(int)*2);
4 MPI_Win win;
5 // Everything above: Negligible ThreadSanitizer impact
6 __tsan_start_racedetect();
7 MPI_Win_allocate(&buf, &win);
8 MPI_Win_fence(win);
9 if (rank == 0) {
10     test[0] = 42;
11     test[1] = 10;
12     MPI_Put(&test[0], win);
13     printf(test[0]);
14 } else {
15     printf(*buf);
16 }
17 MPI_Win_fence(win);
```

---

pute kernel, each instrumented call will return early as all race detection is disabled. Only when the first MPI RMA window be created will the race detection be switched on, and the instrumented calls become expensive. But the opposite is also possible: A flag to disable the race detection may be added to the annotation interface and added to each RMA window destruction. The earlier mentioned flag is replaced with a counter, and the start / stop calls, instead of directly stopping and starting the race detection, increment and decrement this counter. Now, race detection only runs when this counter is not zero. This allows programs with multiple compute sections with different windows to preserve some of the optimization this approach offers, by only guarding the compute kernels themselves. In such a program where it performs preprocessing and compute one after another multiple times, should the window be destroyed after the compute kernel, each preprocessing section can run fast due to the disabled race detection.

When applied to the example to 2.1 this optimization approach will add a race detection start marker in front of the `MPI_Win_allocate` call. This can be seen in Listing 3.4. Every ThreadSanitizer call before the race detection start marker (while still present!) is no longer performance intensive due to the early return caused by the skipped race detection. Combined with the whitelist, now even the last remaining instrumented accesses to `buf` and `test` variables have only negligible performance impact.

### 3.3. Optimize using Remote Access Types

Both optimizations presented before are applicable to any MPI RMA code. In general, dynamic tools have issues with determining specific code characteristics due to their extremely limited code inspection capabilities; should no debug information be built into the binaries there is very little chance to gather meaningful data of the broader code properties. As such dynamic tools must either be applicable very generally, or need additional sources of data by requiring the use of debug symbols or intermediate steps prior to runtime. Currently, MUST-RMA only requires building the code with ThreadSanitizer instrumentation enabled. But ThreadSanitizer is very 'blunt', and does not consider semantical knowledge when deciding on instrumenting a memory access or not. This is intended, as many tools, including MUST-RMA, require the use of the annotation framework. Should ThreadSanitizer decide not to instrument specific calls due to what it determines as irrelevant, some tools may break due to them requiring the existing instrumentation; especially affecting classification quality through false positives or false negatives.

Our analysis is static and domain specific, meaning we are not limited to runtime information and are able to more aggressively filter instrumentation to specifically cater to detection of MPI RMA issues. And so, we describe the following scenario: As described in Section 2.1, a data race occurs when a writing access and a reading or writing access may happen simultaneously. More specifically then, at least one writing access is required for a data race to occur. Additionally since we are only interested MPI RMA data races, we require that at least one of these accesses must happen by way of an MPI RMA call. An exposed window may be written to with `MPI_Put`, and read from with `MPI_Get` (or their request based alternatives). Using static analysis we are able to determine if only a subset of these calls are actually used. Should only `MPI_Get` be used, we know that for any remote access that it must be reading. As data races only occur in combination with a writing access, and there are no remote writing accesses, for any data race to happen there must instead be a local writing access to the exposed memory buffer. Finally we conclude that in this scenario there is no need to instrument local reading accesses for exposed memory buffers. Note that the local buffers are being written to when performing a remote read. These must still be instrumented.

Now consider the reverse case: Only `MPI_Put` is used, and there are no remote reads. Then, the exposed memory buffer reads do need to be considered, but not the local buffer reads<sup>1</sup>. In total, there are two situations where such an optimization can be applied: Either the remote exclusive read, or remote exclusive write code property must be satisfied for this optimization to work. This will be referred to as the 'remote access type'. Additionally, we require any mixed access (see Table 3.1) to not be used. Should none of these two scenarios hold, define the remote access type to be mixed. Table 3.2 shows an overview of the possible scenarios, and which

---

<sup>1</sup>Assuming no multithreading, there can be no simultaneous access to non-exposed memory.

### 3. Optimization Approaches

	Remote Access Types		
	Exclusive Load	Exclusive Write	Mixed
Window Buffer	✓	×	×
RMA access Buffer	×	✓	×

Table 3.2.: Possibility of omitting local load instrumentation of buffer type depending on remote access type

Listing 3.5: Extended recursive step of whitelist generation for remote access type utilization

---

```

1 recurse(Value v, Type t) -> List<(Value,Type)>:
2 // Change call in main to recurse(v,getType(I))
3 List temp = {(v,t)}
4 for Use u of v:
5     if getType(u) != getType(I)
6         t = Dirty
7     if u is function call and getFunc(u) is defined in code:
8         temp = temp ∪ recurse(getParam(u),t)
9     temp = temp ∪ recurse(getUser(u),t)
10 return temp

```

---

accesses may be omitted depending on the scenario encountered.

Here the limitations of static analysis again require a workaround though. As we cannot inspect memory or the chosen control flow during compile time, we also cannot determine which memory buffer is used for which operation. However, the first approach presented in Section 3.1 generates the whitelist originating from the MPI RMA calls. Essentially, this optimization approach works as a filter on this whitelist. The whitelist contains all accesses to the exposed memory or relevant local memory buffers. For this approach we filter these accesses by either exposed memory or local memory (depending on the remote access type), and remove any local reading accesses for them. The whitelist is built recursively from the MPI RMA calls. And these MPI RMA calls determine the type of buffer exactly. Thus, we amend the algorithm used for the first approach: Instead of just saving the relevant parameter when an MPI RMA call is encountered, also save the type of buffer it generates. Then propagate this type during the recursive step. Listing 3.5 shows an overview of the amended recursive step of the whitelist generation. Should during the recursive step a value be encountered which has already been added to the list, and that value has a different type than the current recursion, mark both values as 'dirty' as we are unable to clearly differentiate between the memory types. Finally, once the amended whitelist has been built, and either the remote exclusive read or remote exclusive write scenario apply, ThreadSanitizer can instrument only the relevant reads. Note that any value marked dirty is always instrumented. Therefore this is again an over approximation in order not to cause false negatives during race detection.

Listing 3.6: Extended Whitelist and delayed race detection applied to Listing 2.1. Window Buffers are underlined, Local RMA buffers are dashed.

---

```

1 // Analysis determines code as remote exclusive write
2 MPI_Comm_rank(&rank);
3 int* buf;
4 int* test = malloc(sizeof(int)*2);
5 MPI_Win win;
6 // Everything above: Negligible ThreadSanitizer impact
7 __tsan_start_racedetect();
8 MPI_Win_allocate(&buf, &win);
9 MPI_Win_fence(win);
10 if (rank == 0) {
11     test[0] = 42;
12     test[1] = 10;
13     MPI_Put(&test[0], win);
14     printf(test[0]);
15     // Read in printf is marked in whitelist,
16     // but not instrumented in ThreadSanitizer
17     // as code is remote exclusive write
18 } else {
19     printf(*buf);
20 }
21 MPI_Win_fence(win);

```

---

Consider the example given in Section 2.1. Applying the modified whitelist results in Listing 3.6. Here, only `MPI_Put` is used. As such, it qualifies for the remote exclusive write scenario. In the IR excerpt given in Listing 2.3 we can see that the first element of the local `test` buffer is used for an RMA write, and subsequently the value is output. This read is instrumented, and is not detected for filtering by the first approach; the value pointer is used for RMA operations after all and thus included in the whitelist. Using the approach presented here the whitelist also saves the type of the RMA operation, which is writing *to the target* in this case. In other words, the *local* value is only ever read for the RMA operation. As the code has the remote exclusive write property, we can also safely assume that the buffer is only ever read for remote operations, and that a remote write to the local buffer will not happen. Thus, any reading access does not have to be instrumented. Therefore the present instrumentation of the load instruction on value `%29`, which houses the first element of the local buffer `test` in the `printf` call, can be omitted without compromising race detection accuracy.



## 4. Implementation

All three optimization methods presented have been implemented and may now be used in practice, though with some limitations. Additionally, challenges with technical limitations required changes to the algorithms presented in Chapter 3. The implementation created as part of this thesis also aims to be fully backwards compatible to the original ThreadSanitizer implementation, which also caused some complications. This chapter therefore deals with the practical application of the presented optimization methods as well as their usability.

As for the analysis itself, it is built on top of the LLVM Framework at the current in development version 16 as a 'Module Analysis'. It is built in-tree, meaning the entire framework, including the `clang` compiler frontend, must be compiled for the pass to be available. The framework is currently undergoing a transition regarding the pass manager, which is responsible for deciding on running compile passes or not. Our analysis is built with compatibility for the new pass manager due to LLVM 16 considering the old pass manager deprecated.

Sections 4.1 and 4.2 describe the implementation of the basic whitelist optimization method, with Section 4.2 being especially important for usability concerns. The implementation of the delayed race detection is described in detail in Section 4.3. As this approach requires a modified ThreadSanitizer runtime it also includes information on backwards compatibility. Finally, Section 4.4 introduces the required modifications of the first approach as well as the general code analysis required to determine the remote access type (see Section 3.3).

### 4.1. Basic ThreadSanitizer Whitelist

As the analysis is a module analysis, it has access to one entire IR file at a time. This allows the algorithm to iterate over each function definition in this file, and then iterate over each instruction used within. Iterating this way not only returns the defined functions, but also those that are only declared. After all, the linking step has not yet been performed and thus all external functions have yet to be resolved. Should one of these functions be encountered, the implementation skips them. Finding data races while external functions use RMA buffers is considered out of scope for this thesis. Each instruction is then inspected; if it is a function call then the function name called is compared to a map built into the analysis

## 4. Implementation

Listing 4.1: Workqueue version of Listing 3.2

---

```
1 recurse(Value v, Type t) -> List<Value>:
2 List queue = {v}
3 List newQueue = {}
4 List whitelist = {}
5 while queue not empty:
6     for Value v in queue:
7         whitelist = whitelist  $\cup$  v
8         for Use u of v:
9             if u is function call and getFunc(u) is defined:
10                newQueue = newQueue  $\cup$  getParam(u)
11                newQueue = newQueue  $\cup$  getUser(u)
12        queue = newQueue.copy()
13        newQueue.clear()
14 return whitelist
```

---

itself. This map contains every relevant function (see Table 3.1), as well as the corresponding parameter index containing the buffer<sup>1</sup>. After the buffer parameter index is retrieved from the mapping, the recursive step is called. Finally, once that step completes, the algorithm loops until all functions have been analyzed. Once the main loop body (analogous to Listing 3.1) finishes, the list containing all marked values and instructions is designated as the result of the analysis. Should any other pass invoke the analysis, this list will be returned and cached for any other pass requiring the same (done automatically by LLVM).

While the main algorithm loop (Listing 3.1) has not changed a lot from theory to implementation, this is not the case for the recursive step (Listing 3.2). In fact, the entire algorithm structure was changed to be non-recursive. Recursing for each value quickly lead to stack overflow issues, which required this change. Instead, the algorithm now utilizes a workqueue concept in order to generate the list of connected values for the RMA buffers, a pseudocode version of which can be seen in Listing 4.1. The workqueue is initialized with the relevant parameter of the MPI call, and every element of the workqueue is added to the result whitelist. Any values which would have caused a recursive call are added to a secondary 'new value' list. The algorithm then copies the 'new value' list to the workqueue and clears it. Finally, the algorithm loops until the workqueue is empty, meaning no values were added to the 'new value' list.

Unlike the algorithm presented in 3.1, this loop has an upper limit. Iterating over each connected value causes massive performance issues due to the size of any IR code. To mitigate this, the workqueue is only looped through a certain number of times. As this workqueue works analogous to the previous recursive approach, this is called the 'depth'. During testing, it has become clear that a depth of 6

---

<sup>1</sup>MPI\_Get\_accumulate uses two buffers. This is currently not considered in our implementation, though easy to add.

Listing 4.2: MPI RMA data race, undetectable with whitelist optimization applied

---

```

1  int main() {
2      MPI_Comm_rank(&rank);
3      int* buf;
4      int* test = malloc(sizeof(int)*2);
5      MPI_Win win;
6      MPI_Win_allocate(&buf, &win);
7      MPI_Win_fence(win);
8      if (rank == 0) {
9          test[0] = 42;
10         test[1] = 10;
11         MPI_Put(&test[0], win);
12         printf(test[0]);
13     } else {
14         f(*buf);
15     }
16     MPI_Win_fence(win);
17 }
18
19 void f(x) { g(x); }
20 void g(x) { h(x); }
21 void h(x) { i(x); }
22 void i(x) { j(x); }
23 void j(x) { k(x); }
24 void k(x) { printf(x); }

```

---

provided more than sufficient results during the marking phase. For example, on a C++ code the algorithm was able to correctly identify one of the class attributes being an exposed window buffer, and marked an instruction more than 500 lines (of C code, not IR!) later across function boundaries<sup>2</sup>. Setting the depth to very high values was also tested in order to more precisely understand the effects of this limiter. On C code this changed very little, as it is inherently quite linear. The addition of class structures in C++ however caused severe performance degradation, as once a class function is marked it is also highly likely that this class function is used across the codebase in different contexts. As we are using a static analysis we cannot differentiate between different class objects, causing marked values in functions using unrelated objects of the same type. In theory, the addition of this limiter may cause false negatives due to insufficient instrumentation. Listing 4.2 provides an example one such case, where one of the offending instructions, the `printf` call, is too deeply nested to detect. The workqueue generation will stop too early due to the depth limiting. Moreover such a test case can be built for any depth. However, in practice no test case failed due to this issue outside artificial test cases aimed to exploit this limitation specifically.

---

<sup>2</sup>RMA adaption of miniMD [3], `comm.cpp`, line 590 is instrumented while the buffer is created in line 59.

#### 4. Implementation

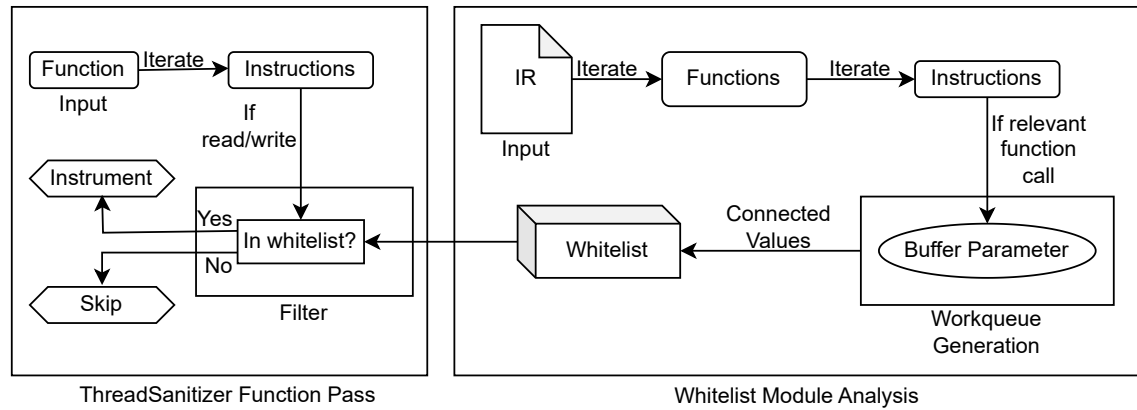


Figure 4.1.: Overview of the whitelist optimization approach implementation.

The loop body of the recursive step has also changed compared to the original approach. Instead of marking all uses, the analysis also considers filtering using alias analysis. The LLVM framework provides basic alias analysis, which is used here. Should a use not be a possible alias (the alias analysis returns `NoAlias`), it is not marked. For any other returned value (`MustAlias`, `PartialAlias`, `MayAlias`) they continue to be marked. This means that only the absolute case is excluded, meaning this can not lead to an accuracy loss. However, the alias analysis is always intra-procedural, which means that as soon as function boundaries are crossed the filtering becomes ineffective.

Outside the uses of an RMA buffer some specific generators of the value are also marked. This is the case for trivial aliasing such as a pointer to pointer load, as well as the `getelementptr` IR instruction. The `getelementptr` instruction returns a pointer to an offset from a pointer, which is often used for array indexing, but also for the implicit indexing present for any pointer dereferencing. This allows the analysis to make sure that it is always the entire buffer that is marked, and not only a temporary variable used specifically for an RMA operation.

Creating the analysis is not enough, as ThreadSanitizer does not use it. Therefore, modification of the ThreadSanitizer compile time pass is required. Internally, the ThreadSanitizer compile time pass is split into two parts, a module pass and a function pass. Only the function pass is relevant to the instrumentation, the module pass only sets up for the function pass. Figure 4.1 shows how the function pass utilizes the module analysis described earlier. However, due to a limitation of the LLVM framework, function passes may not call a module pass. Instead they may only retrieve the cached result if present. To solve this, the ThreadSanitizer module pass was modified to call the analysis once. This makes sure that the cached result is present for the function pass. Finally, the ThreadSanitizer function pass was modified to skip instrumentation of values and instructions not present in the result list generated by our analysis.

In order to use the analysis a flag `-fsanitize-thread-whitelist` was added to

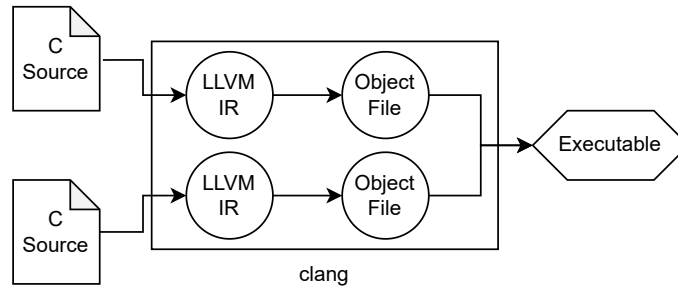


Figure 4.2.: Default clang compilation pipeline

the clang frontend which causes the whitelist analysis to run. The changes to ThreadSanitizer are also backward compatible, and omitting this flag allows for the normal instrumentation as before.

## 4.2. Interprocedural and Cross-Translation-Unit Support

Making the analysis interprocedural is in theory easily doable by inspecting function calls, and continuing the 'recursive' step from inside the function definition. This is also what the implementation does: When a function call is encountered it checks if it is defined or only declared, and, should it be defined *in the IR module*, the relevant parameter of the function definition is marked and added to the workqueue.

A significant issue however is that functions are not always defined in the IR module, even though they are part of the source code. This is due to the way the clang compiler works. Throughout the course of Chapter 3, it was always assumed that the entire source code is available for analysis. But only link time optimization (LTO) passes actually have the entire source available. The analysis written as part of this thesis is a normal compile time analysis, and thus only has access to one module at a time. One module generally corresponds to one C translation unit (TU). clang internally generates the IR for each translation unit, and then runs the analysis, transformation and optimization passes. Then, clang links the transformed IR together, runs the LTO passes and finally creates the executable or library binary. Figure 4.2 provides an overview of the default clang compilation pipeline.

This breaks interprocedural support, as windows may be created in one module and used in another. It is also especially problematic for MPI RMA, as the target rank does not have to participate in any synchronization calls. The target may, after window creation, run functions from a different module with absolutely no MPI or MPI RMA functions used, which would cause the whitelist analysis to not even

## 4. Implementation

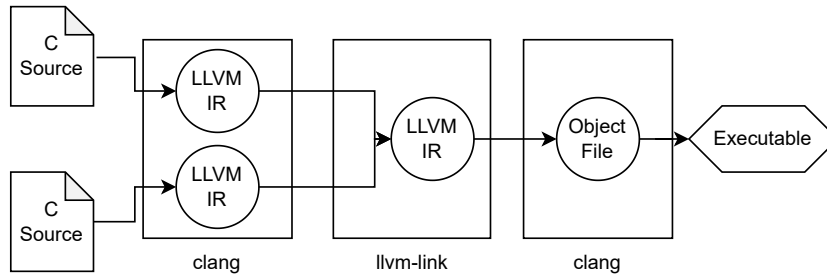


Figure 4.3.: Modified clang compilation pipeline for fixed whitelist support

mark a single value. It is therefore vital to support inter-TU analysis, as without the practical applicability suffers heavily.

Modifying the analysis to run as an LTO pass is possible, but hard due to limited documentation. Even then, the bigger issue is ThreadSanitizer, which would also have to be converted; running the analysis after ThreadSanitizer has already performed the instrumentation is fruitless. Not only would this be a big change to an already established tool, it is also hard to do due to the way the general sanitizer infrastructure is deeply entrenched within the LLVM framework. Instead, the workaround is to manually separate the compilation process into multiple phases. The modified compilation process is shown in Figure 4.2. In the first phase `clang` is used to output LLVM IR for each translation unit, with all passes disabled. Then, `llvm-link` is used to link together all IR files generated. This outputs a single monolithic IR file housing the entire code. Finally, `clang` continues compilation on that file, including any applied passes; especially ThreadSanitizer and our analysis pass. This achieves what is essentially a manual link time optimization, theoretically fixing cross-TU support.

One example of how entrenched the sanitizer infrastructure is in LLVM causes issues here. When the `-fsanitize=thread` flag is used, the ThreadSanitizer compile time pass is enabled and the `clang` compiler is instructed to include the ThreadSanitizer runtime during linking. However, it performs an additional step, namely adding the `sanitize_thread` attribute to all functions in the IR. This step is skipped when the input to `clang` is already an IR file, as is the case with our workaround. To fix this, the analysis is amended. After generating the whitelist, the function definitions in which the values are contained are fetched. Then, the `sanitize_thread` attribute is added to all affected functions. This does not break race detection as, should a function not contain any marked values, no instrumentation would have been applied anyway. It also implicitly provides a way to fully disable instrumentation in irrelevant functions, as normally ThreadSanitizer would still add a preamble, thus helping in optimization (though only minimally).

### 4.3. Delayed Race Detection Start

The implementation of the delayed race detection is significantly less complex than the whitelist, but requires changes to the ThreadSanitizer runtime. The annotation interface of ThreadSanitizer was extended with the `__tsan_start_racedetect` and `__tsan_stop_racedetect` functions, starting and stopping race detection respectively. This is done by adding a global flag within the runtime indicating whether to run race detection, which these function set or clear. When it is not set any call attempting to update shadow memory from a read or write operation are returned from early. As more than one process could call this function at a time, in order to avoid data races in the race detector, the flag is always set and read in atomic operations. Alternatively, `AnnotateIgnoreReads{Begin,End}` and `AnnotateIgnoreWrites{Begin,End}` could be used, as is done in ARCHER [10], a data race detector for OpenMP programs. The evaluation of their use is future work.

These new functions must now be added as compile time instrumentation. A new LLVM function pass was written which simply adds the `__tsan_start_racedetect` function prior to each `MPI_Win_create` or `MPI_Win_allocate` call. Usage of a new pass was necessary as this operation has no return value; an LLVM analysis always returns a result (such as the whitelist) but this optimization approach is just an instrumentation. This pass currently does not add the `__tsan_stop_racedetect` function to `MPI_Win_free` calls. Currently, the runtime only operates on the flag instead of counting active windows as presented in Section 3.2. As both functions were added though, they may be utilized by the MUST-RMA runtime directly in the future to the same effect.

While this approach works using only the changes mentioned, this breaks backwards compatibility with default ThreadSanitizer usage. Therefore, an additional runtime flag is added. ThreadSanitizer already has an options interface using the `TSAN_OPTIONS` environment variable. In order to preserve backwards compatibility, another option `check_races_without_startracedetect` has been added that can be set using the environment variable. Setting this flag allows running ThreadSanitizer as originally intended, and clearing it will cause race detection not to run until `__tsan_start_racedetect` is encountered. Additionally, this flag is set by default, meaning no changes are necessary for existing code. Running with delayed race detection thus requires the environment variable to override this default behavior using `check_races_without_startracedetect=0`.

## 4.4. Extended Whitelist for Remote Access Type utilization

As the last optimization approach presented in this thesis is based on the first whitelist approach, it was most easily implemented by extending the original approach. Therefore, this optimization approach does not require an additional pass like the approaches presented before.

The analysis now does not return a list of values and instructions, but a list of pairs of values and their deduced type. The type is defined as either `RemoteBuf` when the value is exposed at any point, `WriteBuf` if it is used in an `MPI_Put` call, `ReadBuf` if it is used in an `MPI_Get` call, and `DirtyBuf` if it is used in multiple different RMA calls or the analysis was unable to determine the type unambiguously. Note that what happens to the value is the opposite of what the RMA call does to the window, so a value with type `ReadBuf` is *written to* using RMA.

When the whitelist generator encounters a relevant RMA call and marks the buffer parameter for the workqueue generation, it now performs an additional lookup in the parameter index map. Instead of saving a one to one mapping from the function name to the index, it now maps from the function name to a pair, saving both the index as well as the type of buffer it includes. For example, the entry for `MPI_Put` looks like `{"MPI_Put": (0, Type::WriteBuf)}` as the first parameter contains the buffer, and the type is `WriteBuf`. The type information is also passed along to workqueue generation.

When the workqueue is filled, each added value or instruction inherits the type of the value being looked at in the current iteration. Should a value to be added already be in the workqueue list, if the types are equal it is simply skipped. If not, both the original and the new values' type are set to `DirtyBuf` and the new value is skipped. When filling the whitelist, the same is done. Skipping the values does not hinder accuracy by itself. If a value already is in the workqueue, and another with different type is to be added, just iterating over the first will have the same result as doing it for both as we do not differentiate which values to add by the type of buffer inspected. However, the depth limiting can cause issues here. The depth limiting could cause for example a value with type `RemoteBuf` to be discovered first at depth 4 from an `MPI_Win_create`, and the same value to be discovered at depth 5 from an `MPI_Put`. As only one step is left (assuming the default maximum depth of 6), the workqueue will not mark all values which originated from the `MPI_Win_create` as dirty but only those one step closer at depth 3. Listing 4.3 is an example of such a case. The `MPI_Put` is only 4 function calls deep, but both the dereferencing and array indexing may cost a step as well. Thus, when building the whitelist from top down from the `MPI_Win_allocate`, `*buf[0]` is marked correctly. However when marking the parameter `x` in function `i` and checking the generators for aliasing, only `*buf[0]` is marked dirty as the depth limiting prohibits continuing on to `buf`. Therefore,

Listing 4.3: MPI RMA data race, undetectable with *extended* whitelist optimization applied

---

```

1  int main() {
2      MPI_Comm_rank(&rank);
3      int* buf;
4      int* test = malloc(sizeof(int)*2);
5      MPI_Win win;
6      MPI_Win_allocate(&buf, &win);
7      MPI_Win_fence(win);
8      if (rank == 0) {
9          f(*buf[0]);
10     } else {
11         printf(*buf[0]);
12     }
13     MPI_Win_fence(win);
14 }
15
16 void f(x) { g(x); }
17 void g(x) { h(x); }
18 void h(x) { i(x); }
19 void i(x) { MPI_Put(x, win); }

```

---

while all values are contained in the whitelist, the type of `buf` is mistakenly not marked as dirty.

Another issue is transformation of variables and pointer aliasing. During normal whitelist generation, only the uses (as well as trivial aliasing and `getelementptr` pointers) are traversed. However, operations on the pointer variable in the IR could transform it, where only the transformed version is used for mixed purposes, marking it as dirty. For the extended whitelist to work both the original as well as the transformed version must be marked. Recurring function calls could allow the 'later' transformed variable to be used during the 'earlier' parts of the IR, causing data races. This makes marking both the original and the derived version as dirty important, to not allow ThreadSanitizer to skip instrumentation of one or the other. A naive way to fix this is to also mark generators of a value during workqueue generation. This would however mark almost all values, defeating the point of the optimization. It can be improved though: Instead of marking all generators, only 'mark' those that are already in the resulting whitelist, and do not allow new values to be added to the result whitelist from this value on. This causes the algorithm to still consider generators when setting the types, but no longer leads to the high amount of instrumented values. As this was unnecessary on the codes tested, it was left out of the final implementation. Instead a manual searching of value generators was sufficient, more specifically the instrumentation of pointer to pointer loads (see Section 4.1) solved this issue on all test cases checked.

Determining the remote access type of the entire codebase is done by utilizing the

#### 4. Implementation

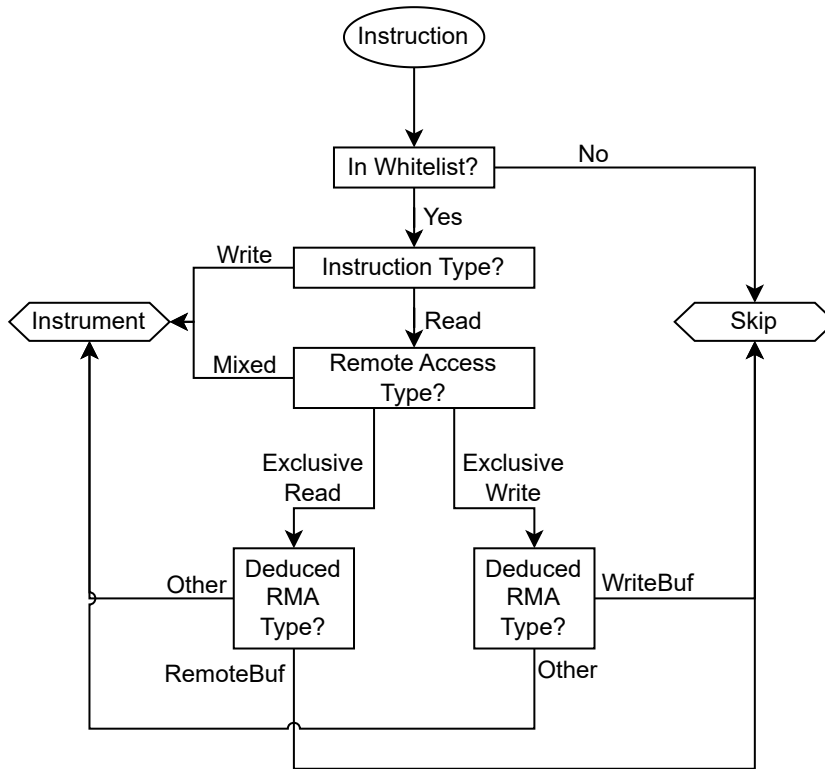


Figure 4.4.: Decision Tree of ThreadSanitizers instrumentation filtering using the extended whitelist.

fact that this approach is extending the already presented module analysis, and with it also inherits the monolithic IR file created by the modified compilation pipeline. The analysis defines two flags representing the remote exclusive read and remote exclusive write respectively. Then, these are set if the conflicting functions are not declared in the IR. Using them would require them to be linked, which can only happen when they were declared, allowing this approach to work. If both are not set it indicates the mixed remote access scenario. These two flags are added to the result, which previously was only the whitelist.

ThreadSanitizer can now inspect each value type as well as the remote access type of the codebase. Using this information, it can now skip the load accesses of the respective value type depending on the remote access type as described in Section 3.3 in addition to skipping values not present in the whitelist at all. Figure 4.4 provides an overview of the logic used when deciding to instrument or skip instrumentation of instructions during the ThreadSanitizer compile time pass, replacing the filter shown in Figure 4.1.

## 5. Evaluation

While the theoretical impact on both classification quality and potential speedup was already discussed, the actual effect has yet to be determined. Thus, this chapter deals with the results observed on a multitude of test cases measuring both the speedup and detection accuracy when applying the static optimization analyses presented in Chapters 3 and 4.

Section 5.1 provides an overview of the testing environment as well as notes on the reproducibility of the results presented in the subsequent sections. Classification quality is discussed in Section 5.2 with a focus on possible false negatives. Section 5.3 presents the speedup measured on select MPI RMA benchmarks using each optimization approach as well as the scaling behavior for different rank counts. Finally, Section 5.4 provides an overview of the results obtained, and discusses the speed and accuracy tradeoff when utilizing each optimization approach.

### 5.1. Experiment Setup

Both the classification quality and overhead study measurements were taken using the JUBE [11] benchmarking environment. JUBE generates workpackages for every step of the compilation and execution process, for each test case configuration. The results are reliable, in that each step is performed in exactly the same way when repeated, given that the JUBE benchmark script was not changed in that time. The scripts used for both the classification quality and overhead benchmarks are provided as part of this thesis.

While the steps are guaranteed to be reproducible using the benchmarking environment used, different systems may still execute each step differently. This may be caused by different compilers used, C libraries, MPI implementations or other possible variations in an end user system compared to the testing system used for development. In order to mitigate these differences, a Docker container is used to run the classification quality benchmark, though built and run using Podman. This provides reproducible environments for the classification quality benchmark to run by fixing all package versions as well as providing a sandbox from the host system, mitigating possible influence thereof on the test results. The container used for

## 5. Evaluation

this thesis is built on an Ubuntu 22.04 system utilizing LLVM 13 for the bootstrap compiler<sup>1</sup> and MPICH as the MPI implementation.

For the overhead study a high performance system is needed, and such a container may be detrimental. Additionally, HPC systems usually work on a module system, allowing for reproducibility even without the use of containers by loading the version-locked modules before running JUBE. The performance evaluation was therefore run directly on the HPC system, without using a container sandbox. Simulations were performed with computing resources granted by RWTH Aachen University under project `thes1341`. The system used is the RWTH CLAIX-2018 cluster [18], using LLVM 13 as the bootstrap compiler and Intel MPI as the MPI implementation. The overhead study was run on the Rocky Linux 8 portion of the cluster. Each node of the cluster utilizes two Intel Xeon Platinum 8160 CPUs with 24 cores each, and the cluster has  $\approx 1250$  nodes in total connected using Intel Omni-Path. For the purposes of the overhead evaluation a maximum of 8 nodes will be used per test case. MUST execution fully utilizes each node, meaning 24 application and 24 tool processes running on each node. For non-MUST execution only the 24 application processes are used, leaving the other spots vacant.

### 5.2. Effect on Classification Quality

As mentioned in Chapter 1, static analysis tools often have low detection accuracy. While improving performance is the main goal of this thesis, it is important to make sure this does not come at the cost of the MUST detection accuracy. If the accuracy of the dynamic tool is affected significantly, there is no longer a need to use a dynamic tool. After all, in the general case a purely static tool will run much faster than any dynamic tool, even with a multitude of optimizations applied. The goal of this thesis is to *preserve* the accuracy gain of a dynamic tool, and enhance its runtime performance. Thus, in the following the effect on classification quality is measured with regard to the optimization implementations.

The test cases on classification quality are separated into two classes: single-file, where a single C source file is compiled, and multi-file, where compilation consists of at least two C source files. This is due to the different compilation approach used for each (see Section 4.2). In total, 3 multi-file and 93 single file test cases were evaluated for a sum of 96. Many of the single file test cases were sourced from the MPI Bugs Initiative [9] as well as the existing MUST [16] test suite (Prefix `EXTERN_` in repository). In addition to these, 11 single file and all multi-file test cases are new, written explicitly to test the analysis presented. Of all test cases, 34 are negative test cases exhibiting no data races, and 61 are positive containing at least one data

---

<sup>1</sup>This compiler is only used to compile MUST and the modified LLVM repository containing the optimization analysis implementation. All test cases are compiled using the modified LLVM suite.

race. All but 18 test cases run on two ranks, the other on 3. The test cases were run on all possible configurations of the static optimization analysis in order to make sure all accuracy regressions are detected. These configurations are:

1. Immediate race detection. All values instrumented (Base case).
2. Immediate race detection. Whitelist is used.
3. Immediate race detection. Extended Whitelist with type utilization is used.
4. Delayed race detection. All values instrumented.
5. Delayed race detection. Whitelist is used.
6. Delayed race detection. Extended Whitelist with type utilization is used.

This produces 6 configurations per test case.

Initially, running the test suite revealed one regression caused by a programming error when detecting remote access types. After fixing this no other regressions were detected. Some external test cases however still failed, but were deemed to be unrelated to the optimization analysis. 2 test cases were expected to fail for MUST without the optimization applied already, and are therefore not regressions. Finally, 7 test cases failed unexpectedly as false negatives. However, rerunning them allowed MUST to detect these issues without a problem. As the static analysis produces consistent results given the same code, we conclude that these failures cannot have been caused by it. Thus, after fixing the aforementioned issue, there were no regressions in external test cases.

While the static analysis caused no issues with the external test cases, some internal test cases failed due to the static analysis applied at compile time. Of the 10 single file test cases, 4 failed due to false negatives. Each of these was explicitly constructed to break due to the whitelist or extended whitelist optimization. The first of these is `test_expfailwhitelist_nok` which causes issues due to deeply nesting a reading access in multiple function calls (analogous to Listing 4.2). To reach the conflicting reading access a higher maximum depth value is required, and, setting a higher does indeed fix this test cases. The second is `test_expfailwhitelistext_nok`, which fails due to a similar but not the same issue (analogous to Listing 4.3). While all relevant values and instructions are instrumented, their type information is not propagated fully. This causes some values to not be marked dirty while they should be, causing missing instrumentation of the load accesses. Again, setting a higher depth value may fix this issue.

Another expected failure is `test_expfailparam_nok`, caused by pointer aliasing and function parameters. While interprocedural support is present, this breaks down if pointer aliasing is created at the same time. Consider a function of the form `f(int** x, int** y) = { *y = *x }`, as is the function `aliasgenerator` in Listing 5.1. If this function is called at any point with the first parameter being a whitelisted value, though the function body is instrumented, the parameter value `y`

## 5. Evaluation

Listing 5.1: MPI RMA data race, undetectable due to missing backpropagation of function parameter aliases.

---

```
1 int main() {
2     MPI_Comm_rank(&rank);
3     int* buf;
4     int* aliasbuf;
5     int* test = malloc(sizeof(int)*2);
6     MPI_Win win;
7     MPI_Win_allocate(&buf, &win);
8     aliasgenerator(&buf, &aliasbuf);
9     MPI_Win_fence(win);
10    if (rank == 0) {
11        MPI_Put(*buf[0], win);
12    } else {
13        printf(*aliasbuf[0]);
14    }
15    MPI_Win_fence(win);
16 }
17
18 void aliasgenerator(int** x, int** y) {
19     *y = *x;
20 }
```

---

is not within the call origin. Any writing or reading access on `y` may therefore lead to undetectable data races. Listing 5.1 has two variables `buf` and `aliasbuf`, which are set to the same value in the `aliasgenerator` function. As this is not recognized in the calling function, the `printf` call remains uninstrumented. In theory, this may be fixed by backpropagating possible aliasing to the calling function. This can be done by adding a 'origin' marker to parameter values, which stores a list of places the function `f` is called from with a relevant parameter. Then, should another parameter be marked, it is also marked for every origin function in the list. This is however currently not implemented and considered future work. The final expected false negative is caused by function pointers. While most code relies on calling functions directly, some utilize function pointers which may be any function with a fitting signature. As the possible values for a function pointer are not known prior to runtime, this is an inherent limitation of the approach used, and cannot be fixed easily. However, it can be mitigated: On the assumption that the modified compilation process is used, all possible functions called are known. Then, all defined function's signatures are compared to the signature used for the function pointer. All that fit are then instrumented according to the normal whitelist workqueue generation. The implementation of this mitigation is considered future work.

All multi-file test cases ran without issue when the optimization is applied. However, when running without any optimizations applied, the data races are not recognized. This is easily explained by the modified compilation process used (see Figure 4.3). Normally ThreadSanitizer applies the `sanitize_thread` attribute to every function,

but this only happens when the input file is not yet in IR form. As with this modified process ThreadSanitizer is only applied afterward, this attribute is missing and ThreadSanitizer does not instrument the functions<sup>2</sup>. The optimization analysis pass manually adds this attribute and any multi-file test case configuration utilizing it detects data races correctly.

In general, we conclude that the whitelist and extended whitelist optimization do cause accuracy regressions, namely false negatives. We were unable to create a test case that fails due to the delayed race detection, even though in theory there should be no failures caused by it. After all, the race detection is only delayed until the first possible race, not later. Both in theory and in practice no false positives could or were caused by the analysis pass. Additionally, all false negatives caused were in artificial test cases created explicitly to cause false negatives. The limitation caused by the missing propagation of function parameters is however a significant drawback to the whitelist approach, and must be kept in mind when using it.

### 5.3. Overhead Study

The overhead was measured on separate test cases from those used during classification quality evaluation, as those test cases are not computationally intensive enough for worthwhile performance measurements. Instead, four separate test cases were measured: A standard matrix stencil, which performs a 5-point stencil operation on a square matrix using RMA to handle the halo exchange, and transpose code, which utilizes RMA to perform a blockwise transpose operation per process, from the Parallel Research Kernels repository [19]. Additionally the performance of miniMD [3], a molecular dynamics simulator, as well as miniVite [5], a graph community detector, is measured. miniMD, though not native RMA code, was adapted to utilize RMA instead of standard two-sided communication. miniVite has built-in support for RMA communication: There is an `MPI_Accumulate` and `MPI_Put` version; the `MPI_Put` version was chosen for the performance measurements. All performance test cases do not contain data races. The test cases were run with and without MUST for each of the 6 configurations used when measuring the classification quality. Additionally the code was run without any instrumentation as a base measurement, for a total of 13 tested configurations. Finally, to mitigate possible outliers all configurations were run five times, and the execution times presented are the average of these runs.

All performance test cases were run on 2, 4 and 8 nodes for a total of 48, 96 and 192 application ranks respectively. The MUST processes will not be counted throughout this evaluation. The stencil kernel ran on a 20000x20000 matrix for 1000 iterations,

---

<sup>2</sup>Apart from the preamble mentioned in Section 4.2

## 5. Evaluation

and the transpose kernel ran on a 15360x15360 matrix<sup>3</sup> for 500 iterations. miniMD was given the default supplied input file `in.1j.miniMD`, and simulated 8000 time steps. The exact data is irrelevant, as the only relevant result is the time taken for execution. Finally, miniVite was given a road network<sup>4</sup> of the USA for analysis.

All test cases developed some form of speedup compared to normal MUST or ThreadSanitizer execution, with minimal error across all tests. This can be seen on the improved slowdown in Figures 5.1 through 5.4. Here, the `must` and `tsan` prefix denote whether or not MUST was run in addition to ThreadSanitizer instrumentation, and the `_std` and `_opt` suffixes indicate standard or optimized execution respectively. The stencil kernel especially had significant speedup, reaching extremely close to the reference baseline measurements (see Figure 5.1). Though initially unreasonable, inspecting the resulting IR did show that relevant memory locations were instrumented correctly, and data races could still be detected when injected. The reason for the speedup is a difference in programming paradigm compared to the other test cases, in that the stencil kernel has separate communication and computation phases. Especially important is that the computation phase does not have remote access, and is thus not instrumented by the whitelist approach. And while the communication phase is instrumented, it is computationally insignificant. Therefore, stencil is a best case scenario for the whitelist approach, where the hotspot of the code is irrelevant to race detection.

In contrast, Figure 5.4 shows that miniVite had a significantly worse, though still measureable, speedup. The optimizations only speed up the code by about 25%, nowhere near the 4x achieved in the stencil test case. This is expected due to the high communication overhead of miniVite, as well as many exposed buffers being used during computation; it is the opposite result of the stencil code representing a code with relatively unsuitable conditions for a whitelist speedup.

Both transpose (Figure 5.2) and miniMD (Figure 5.3) reduced slowdown by around a factor of 0.6x to 0.5x, meaning a speedup of up to 2x. These seem to be representative measurements for an average run of optimized code in contrast to the outliers mentioned earlier. The slowdown increases with the amount of ranks for the transpose kernel, indicating that the instrumentation of the communication buffers is not insignificant as it was in the stencil code. This does not seem to be the case for miniVite. However, the miniVite execution finishes a lot earlier than the transpose kernel. It is likely that the constant overhead portion of miniVite becomes significant at higher core counts, leading to the appearance of stagnant slowdown.

The whitelist approach worked in these test cases by far the best, offering an average speedup of  $\approx 2x$  (see Figure 5.5), with some outliers such as stencil performing extremely well, and miniVite in which it performed worse. The stencil case is easy

---

<sup>3</sup>The matrix size here is largely decided by the code requiring the matrix order to be divisible by the number of ranks.

<sup>4</sup>`road-road-usa` sourced from [13]

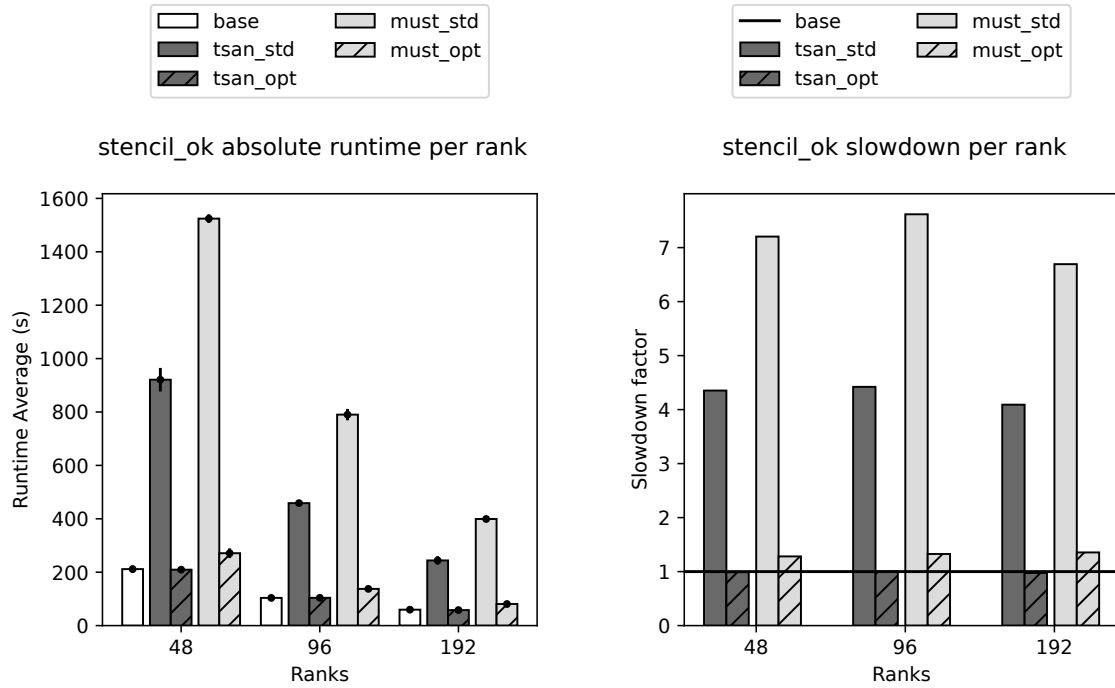


Figure 5.1.: Stencil Kernel performance results.  
 Left: Absolute runtime. Right: Slowdown for each scenario.  
 The standard deviation is used as the error.

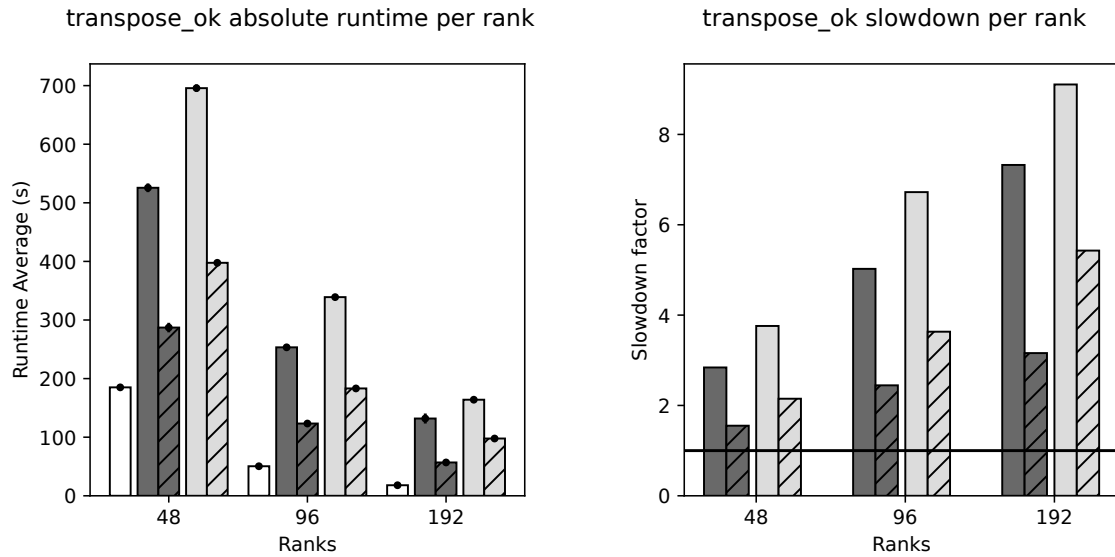


Figure 5.2.: Transpose Kernel performance results.

## 5. Evaluation

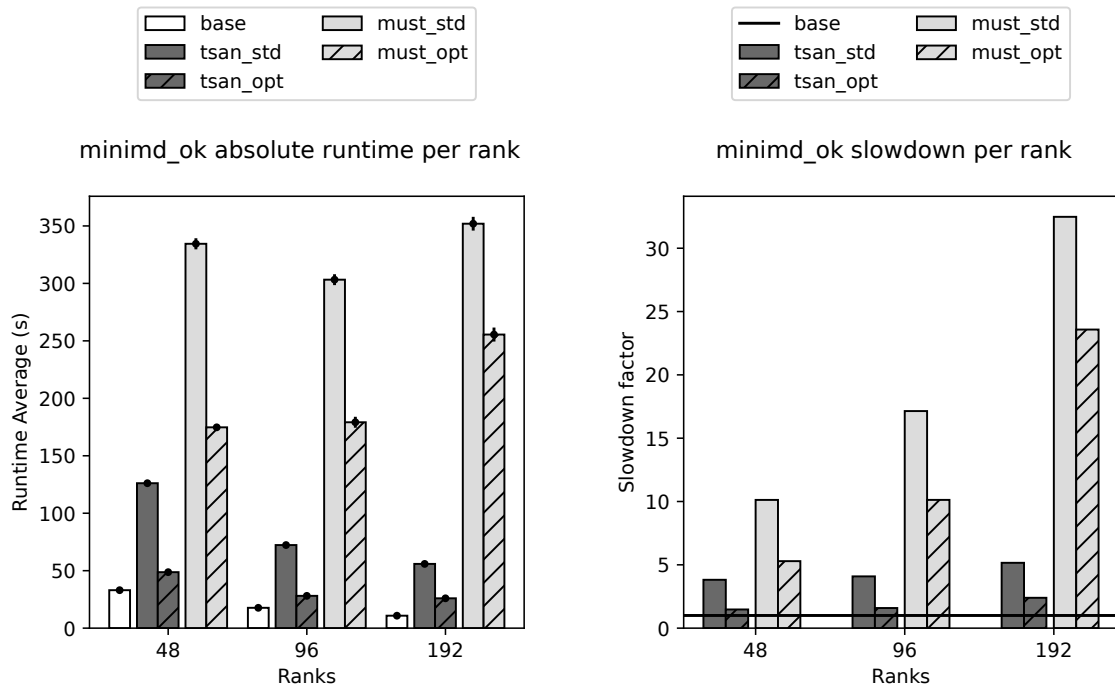


Figure 5.3.: miniMD performance results.

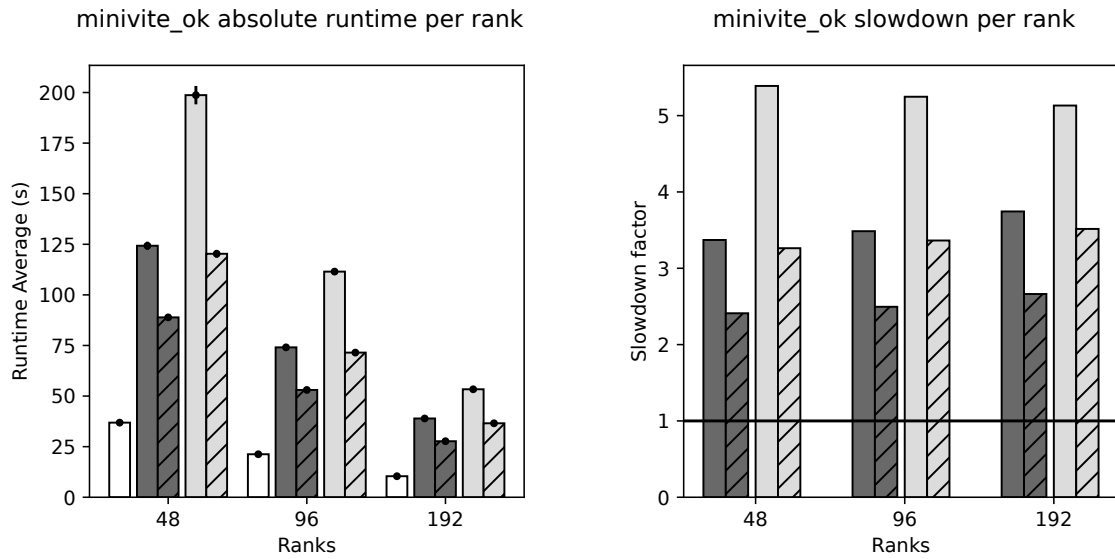


Figure 5.4.: miniVite performance results.

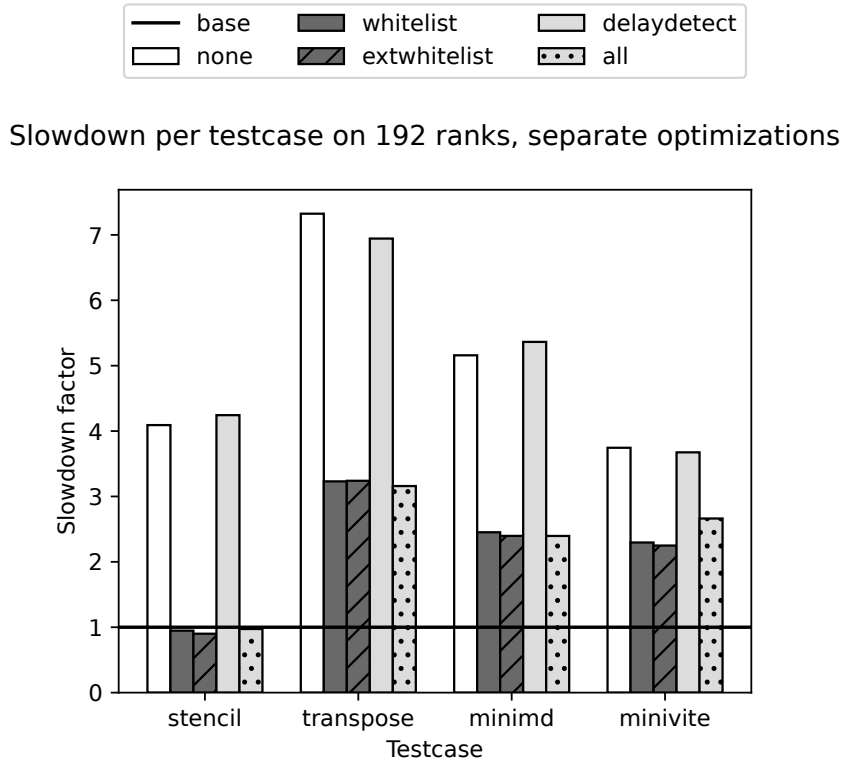


Figure 5.5.: Slowdown per optimization method for each test case. All runs here without MUST.

to explain; due to the mentioned clear separation of compute and communicate phases the whitelist works at peak effectiveness. Transpose and miniMD represent the average speedup gain possible with the whitelist approach. Both perform local computation and afterwards sync their changes, though not as cleanly separated as stencil. miniVite however has a high communication overhead. The possible gains for omitting some local ThreadSanitizer instrumentation is thus lower than possible on other performance test cases.

While the delayed race detection did not perform well in the graphs presented, it is not useless; the graphs are measured on kernel time only. The delayed race detection performs best on code with high setup costs. With all test cases omitting the setup phase on time measurement it is no surprise that the speedup gain is not present. However, when measuring the entire program lifetime it is clear that only miniVite showed a clear benefit from the use of delayed race detection: It reduced slowdown from around 3.6x to 3.3x. This is unsurprising as, compared to the other test cases, miniVite does not run on simulated data but instead real data read in during the programs setup phase prior to kernel execution. Reading a  $\approx 1$ GB file is not easily done with ThreadSanitizer weighing down on every memory access, and thus the delayed race detection can provide a speedup. As this is still not a dominant part of execution, it still cannot reach the speedup gain from the whitelist.

## 5. Evaluation

The remote access type utilization was ineffective. The stencil hotspot is already largely uninstrumented by the base whitelist, and additional filtering would only provide marginal improvements in any case. While transpose is remote write only, the local RMA buffers are written to often, rendering the filtering ineffective. miniMD is remote exclusive read, allowing skipping of window read instrumentation. However, only an insignificant portion of the whitelist were window reads, meaning no speedup gain. Additionally, the code hotspot is mixed with the RMA communication, causing the instrumentation that was present to be significant. miniVite is remote exclusive write, and local RMA buffer reads do not have to be instrumented. Here the opposite from miniMD is the case, in that most whitelist entries are window buffers. Thus, there again is no speedup gained from this approach<sup>5</sup>.

While combining all approaches is the most consistent way of getting the best possible speedup, sometimes they also slow down execution (Figure 5.5) The whitelist approach for example may lead to unwanted side effects in the ThreadSanitizer runtime; it can move shadow memory writes closer to the hotspot of the program, causing slowdown in critical areas. This can happen if the first instrumented write to a memory region happens inside the code kernel, as the initialization phase may not be whitelisted. Early versions of the delayed race detection actually consistently slowed down execution due to side effects. Now some slowdown is still present due to access of the atomic race detection enablement flag. Thus, achieving the best optimization is program-dependent and, in the case of the delayed race detection, very input-dependent.

## 5.4. Discussion

The results obtained in Section 5.2 made clear that some classification quality loss is encountered when using some of the presented optimizations: The whitelist and extended whitelist optimization introduce new limitations on the detection capability of the MUST-RMA race detector. With the delayed race detection approach unaffected, it provides an easy to understand and use improvement over current methods, but not particularly effective on code with insignificant setup costs. Even then, with no accuracy loss there is no downside to utilizing this optimization approach. Though some code is slowed down by the additional synchronization of the runtime flag, it has been measured to be insignificant. The possible speedup gain on code with high setup costs, such as the measured miniVite, may more often than not be worth the tradeoff. It is important to note that the performance test cases really are just test cases, and real word applications especially for HPC workloads will require great amounts of data to compute. On longer running codes the race detection may also be toggled on and off when the last open window is freed for

---

<sup>5</sup>More specifically, the speed differences are well within the error margin.

additional speed gain, though, as mentioned in Chapter 4.3, this is currently not implemented (though the interface calls were added for any external tools to use).

Should more speed be required, or the program is unsuitable for the delayed race detection, the whitelist approach can be used. For codes programmed with separate communication and work cycles, the whitelist approach is especially useful, providing significantly faster execution times. As it is fully static and does not rely on the ThreadSanitizer runtime the results are also very predictable. Additionally, for long-running codes or when debugging a data race only occurring very late into the program lifetime, the whitelist approach allows for higher than standard debugging performance whereas the delayed detection becomes ineffective as soon as one MPI window is created. Additionally, this approach may also be extended in the future: Currently, the whitelist saves only the memory pointers possibly containing an RMA buffer, and any access to these pointers is instrumented and inspected. However, some codes may overallocate their RMA buffers, exposing more memory than is actually accessed. It may be fruitful to save a possible memory range for each pointer in the whitelist, where only this range can be remotely accessed. Local accesses outside this range can be skipped. Due to difficulty with evaluating these ranges, this extension to the whitelist approach is considered future work.

The extended whitelist approach, while functional, is not worthwhile on the codes tested. With no improved runtime but decreased detection accuracy there is currently no use for this optimization. As all test cases qualify for one of the read or write only remote access type, and this optimization did not improve runtime measurably in any, it is doubtful that many if any codes exist where this optimization could prove useful.

While the detection accuracy is affected when using the whitelist or extended whitelist approach, the accuracy loss is minimal. None of the existing accuracy test cases caught an error in the data race detector with any combination of optimizations applied. Only manually writing test cases to exploit the limitations of these approaches lead to false negatives. Additionally, the limitations regarding function pointers and function parameter backpropagation may be addressed as future work. Especially important is the lack of false positives; we were unable to create a scenario where any of the optimization methods presented caused a false positive.



## 6. Conclusion

MPI RMA provides one-sided communication in contrast to the standard two-sided calls of the MPI specification. Though experiments with RMA show promise of performance improvements compared to two-sided communication, incorrect usage may easily lead to data races. Dynamic analysis tools may help, but come at the cost of significant slowdown. Static tools are cheap to use, but are very inaccurate in comparison.

To mitigate this dilemma, this thesis presents three different optimization techniques to speed up the execution of dynamic analysis tools. The first is the basic whitelist, which tries to exclude irrelevant memory accesses from being inspected by the race detector, saving on computation time. The delayed race detection cuts off most of the initialization time of the code from being inspected by the race detector, allowing the dynamic analysis to only start up once a data race may actually happen. Finally, static analysis may be used in order to determine specific code properties such as the remote access type defined in Section 3.3. This allows for an extension of the whitelist approach, which can work to filter some entries of the whitelist and prevent their instrumentation.

The effects of the static analysis optimizations were evaluated using MUST-RMA, a dynamic race detector for MPI RMA. As MUST-RMA utilizes ThreadSanitizer to perform all local instrumentation, and this being a significant contributor to the slowdown caused when running MUST-RMA, all of the optimizations focus on speeding up this part of the execution process. Thus, they were implemented as compile time passes in the LLVM framework to be able to directly influence the ThreadSanitizer instrumentation and runtime. The choice of the LLVM framework as the implementation basis also provides much flexibility, with all optimizations supporting both C and C++ code including the preservation of interprocedural support as well as the race detection across translation units.

Using these implementations on the performance test case suite created as part of this thesis resulted in halving the slowdown caused by the ThreadSanitizer instrumentation. In the best case, slowdown was reduced from 4x to 1x, making the instrumentation slowdown negligible, while in the worst case slowdown was only reduced from 3.75x to 2.75x. Dissecting the results by optimization method shows that the bulk of the speedup is caused by the whitelist approach; the minimized ThreadSanitizer instrumentation is much faster compared to the default of instrumenting any and all accesses to memory. The delayed race detection did not fare

## 6. Conclusion

as well, but on one specific test case it did reduce slowdown by 0.3x. While not as fast as the runtime with the whitelist approach, it does show that on code with high setup costs, such as in this case reading a large file prior to computation, there are speed gains to be had using the delayed race detection. Finally, the extended whitelist did not allow for any speedups on the test cases used. Though all four test cases qualify for an optimizable remote access type, the lacking performance benefit shows that this optimization method does not provide a use at this time.

The high speedup gain does, in certain cases, come at the cost of classification quality. While the delayed race detection preserves the detection accuracy of the original tool fully, the same cannot be said for the whitelist-based approaches. Both may lead to false negatives, with the most significant reason being the depth limiting used when iteratively generating the whitelist. The depth limiting is also the cause for the extended whitelist performing worse in classification quality than the basic one, as deciding the RMA buffer types in the whitelist requires a higher depth limit than just using the whitelist without filtering. Other than that there are also some issues with pointer aliasing in function parameters, as well as lacking support for function pointers. In all cases it is however important to note that these issues only occurred in highly artificial test cases written explicitly to show the limitations of these approaches, with all externally written test cases unaffected. Thus, while accuracy loss is present, in practice it is thought of to be minimal.

In conclusion, the optimization methods presented may well be used in practice. Their implementation is also general enough to be extended to other programming languages such as Fortran in the future, making the current implementation a basis for possible extensions. The Fortran extension especially only requires adaptation to a different call structure in the LLVM IR, as well as adding the different MPI RMA calls to the list of relevant functions (see Table 3.1). The lack of false positives with any optimization method, combined with the average 2x performance boost allows for more rapid prototyping of bugged code. Faulty programs can be debugged with optimizations applied at first, and, in case the issues were not detected, the code can be checked without their use. Without false positives, should an error already be detected with the optimized version, there is no need to run the code without race detection optimization applied. The limitations regarding classification quality using function parameter aliasing and function pointers may also be addressed, with some theoretical solutions already presented in Section 5.2. Finally, the accuracy loss only occurs on the whitelist-based approaches, which means the delayed race detection can be enabled any time RMA data race detection is done.

# Bibliography

- [1] Tassadit Célia Aitkaci, Marc Sergent, Emmanuelle Saillard, Denis Barthou, and Guillaume Papauré. “Dynamic data race detection for mpi-rma programs”. In: *EuroMPI 2021-European MPI Users’s Group Meeting*. 2021.
- [2] Zhezhe Chen, James Dinan, Zhen Tang, Pavan Balaji, Hua Zhong, Jun Wei, Tao Huang, and Feng Qin. “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*. IEEE. 2014, pp. 499–510.
- [3] PAUL CROZIER and STEVEN PLIMPTON. *miniMD v. 1.0*. Tech. rep. Sandia National Laboratories (SNL), Albuquerque, NM, and Livermore, CA . . . , 2009.
- [4] Robert Gerstenberger, Maciej Besta, and Torsten Hoefer. “Enabling highly-scalable remote memory access programming with MPI-3 one sided”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2013, pp. 1–12.
- [5] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, and Assefaw H Gebremedhin. “miniVite: A graph analytics benchmarking tool for massively parallel systems”. In: *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE. 2018, pp. 51–56.
- [6] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R de Supinski, and Matthias S Müller. “MPI runtime error detection with MUST: advances in deadlock detection”. In: *Scientific Programming* 21.3-4 (2013), pp. 109–121.
- [7] Tobias Hilbrich, Martin Schulz, Bronis R de Supinski, and Matthias S Müller. “MUST: A scalable approach to runtime error detection in MPI programs”. In: *Tools for High Performance Computing 2009: Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*. Springer. 2010, pp. 53–66.
- [8] MPI Bugs Initiative. *Current Results as produced by continuous integration job*. <https://mpibugsinitiative.gitlab.io/MpiBugsInitiative/>. [Online; accessed May 11, 2023].

- [9] Mathieu Laurent, Emmanuelle Saillard, and Martin Quinson. “The MPI bugs initiative: a framework for MPI verification tools evaluation”. In: *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE. 2021, pp. 1–9.
- [10] Gregory L Lee, Dong H Ahn, Ignacio Laguna Peralta, Martin W Schulz, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Joachim Protze, and Simone Atzeni. *Archer*. Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2019.
- [11] Sebastian Lührs, Stephan Graf, Alexander Schnurpfeil, Wolfgang Frings, and Kay Thust. *JUBE—A Flexible, Application-and Platform-Independent Environment for Benchmarking*. Tech. rep. Jülich Supercomputing Center, 2015.
- [12] Yussur Mustafa Oraji. *Test cases and result data for this thesis*. <https://publications.rwth-aachen.de/record/953805>. [Online; accessed May 11, 2023]. DOI: 10.18154/RWTH-2023-02710.
- [13] Ryan A. Rossi and Nesreen K. Ahmed. “The Network Data Repository with Interactive Graph Analytics and Visualization”. In: *AAAI*. 2015. URL: <https://networkrepository.com>.
- [14] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. “PARCOACH: Combining static and dynamic validation of MPI collective communications”. In: *The International Journal of High Performance Computing Applications* 28.4 (2014), pp. 425–434.
- [15] Emmanuelle Saillard, Marc Sergent, Célia Tassadit Ait Kaci, and Denis Barthou. “Static Local Concurrency Errors Detection in MPI-RMA Programs”. In: *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE. 2022, pp. 18–26.
- [16] Simon Schwitanski, Joachim Jenke, Felix Tomski, Christian Terboven, and Matthias S Müller. “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)*. IEEE. 2022, pp. 27–36.
- [17] Konstantin Serebryany and Timur Iskhodzhanov. “ThreadSanitizer: data race detection in practice”. In: *Proceedings of the workshop on binary instrumentation and applications*. 2009, pp. 62–71.
- [18] RWTH Aachen University. *Overview and technical details of compute clusters at RWTH Aachen University*. <https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/article/fbd107191cf14c4b8307f44f545cf68a/>. [Online; accessed May 11, 2023].
- [19] Rob F Van der Wijngaart, Evangelos Georganas, Timothy G Mattson, and Andrew Wissink. “A new parallel research kernel to expand research on dynamic load-balancing capabilities”. In: *High Performance Computing: 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, June 18–22, 2017, Proceedings 32*. Springer. 2017, pp. 256–274.

## A. Appendix

All test cases as well as the runtimes of all performance tests are available at [12] (Publication RWTH-2023-02710), including the tools and scripts required to reproduce these results. Test cases are at `tests/single_file`, `tests/multi_file` and `tests/perf_tests` in the archive, with their results in `tests/eval_data`. This folder also includes the Python script used to generate the graphs used in this thesis.

The JUBE benchmarking environment is required for running the tests cases. Should installing JUBE not be an option, a Docker file building all required programs is supplied as well. Using a Docker image is recommended for ensuring compatibility. This ensures correct compiler and MPI versions. When not using the Docker image, note that OpenMPI especially may cause issues with MUST-RMA.

To reproduce the results presented, first the custom LLVM installation and MUST have to be set up. This is automated in the Docker file. The custom LLVM installation is present in `LLVMPass/llvm-newpass` in the thesis archive, and can be built the same as a normal LLVM installation. However, the build folder is required to be at `LLVMPass/llvm-newpass/build` for the scripts to work correctly. After building LLVM, MUST is built using the supplied `build_must.sh` script present in `LLVMPass/`.

The JUBE scripts can be run with `run_tests.sh` and `run_tests_hpc.sh` depending on whether the scripts are being run on a local machine or HPC system respectively. Only the CLAIX-18 cluster was tested for the HPC script, and it is the only supported platform for it. After finishing the scripts will print the result table as given from JUBE; for the correctness benchmark all configurations and number of errors reported, and for the performance benchmark the runtime average and standard deviation. The working directory for these benchmarks, including the resulting output, is at `tests/jube_correctness_out` and `tests/jube_performance_out` for the correctness and performance benchmarks each.