Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Masterarbeit

Verification of one-sided MPI communication code using static analysis in LLVM

vorgelegt von

Marcel Heing-Becker

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	7004013
Erstgutachter:	Prof. Dr. Thomas Ludwig
Zweitgutachter:	Dr. Michael Kuhn
Betreuer:	Jannek Squar
	Dr. Michael Kuhn

Hamburg, 2019-02-25

Abstract

When implementing a software model using the one-sided communication of MPI, the developer is required to obey certain rules with respect to memory consistency, process synchronization and operation completion in order to avoid data races and undefined behavior. For a non-trivial application, dealing with different synchronization and memory models that are provided by MPI version 3, manual correctness checks may turn out to be cumbersome.

This thesis evaluates the ability to perform a verification of properties specific to the use of one-sided communication of MPI by statically analyzing an LLVM intermediate representation using two different IFDS problem models, powered by a static analysis framework named Phasar.

Contents

1. Introduction

In the domain of *high-performance computing* (HPC), in order to scale software computationally to the dimension of available processing nodes, choosing and embedding an MPI implementation is a method of choice. Historically, the MPI standard grew over a focus on two-sided communication, imposing the handling of communication and synchronization upon the sender and receiver processes for every single interaction. With MPI standardized by version 3.1, an implementation must provide a range of abilities to enable different flavors of one-sided communication across processes for relaxed semantics, most notably by a consolidated concept called *windows*.

As for any *application programming interface* (API), MPI one-sided communication likewise requires the software developer to obey various rules to maintain functionality as expected: MPI defines a set of methods to manage synchronization and communication for windows separately. Failing to order calls to these methods appropriately in both intra- and inter-process manners may result in deadlocks or non-deterministic behavior.

An approach to detect such maloperations and to notify the software developer in advance is a static analysis of the source code. The MPI standard defines its methods and data types for the programming languages C and Fortran; therefore, it also allows C++ code to seamlessly adopt and link to any MPI declaration first-hand. To address the syntactical spectrum of these languages, and to work on arbitrary stages of code transformation and optimization, this thesis presents a static analysis of programs using MPI one-sided communication, implemented on top of the intermediate program representation of the LLVM compiler framework.

Initially, chapter 2 introduces MPI one-sided communication from the perspective of the standard to the extent that is required to understand the challenges being summoned from its use. Chapter 3 will then introduce essential components of LLVM and justify the benefits of performing static code analysis on its intermediate representation. Additionally, a fairly young LLVM static analysis framework called *Phasar* will be presented for its acceleration in the development of the static analysis tool. In chapter 4, the thesis explains how to make use of the IFDS problem solution algorithm – the heart of Phasar – for its analysis purpose, and it provides a set of analyses to cover the property testing of code under the constraints presented by chapter 2. Next, in chapter 5, the implementation is primarily evaluated for concerns of performance and scalability. The results help understanding a second approach that is given by chapter 6, shifting the balance between performance and features. Given a feeling for the topic of this thesis and its challenges, chapter 7 puts related work into the spotlight. Further limitations and ideas are listed by chapter 8 that serves as a navigation for future work on the topic and the particular implementation.

2. MPI one-sided communication

When examining the MPI standard and its chapter on one-sided communication, it turns out to provide different models of inter-process synchronization, serving different communication patterns. In this chapter, all of the window synchronization concepts are being presented by demonstrating their application and by pointing out dangers in case of an incorrect use.

2.1. Prerequisites

For this thesis, the *Message Passing Interface* (MPI) is subject to the latest standardized version only, version 3.1 at the time of writing. In addition, this thesis strictly refers to the reference definitions by [For15], ignoring any kind of implementation-specific hints, optimizations and customizations that may be defined in documents of individual MPI implementations. When referring to interfaces of MPI, this thesis sticks to the variant of the C language.

2.2. Two-sided communication

To better understand the niche that one-sided communication attempts to fill, an overview of two-sided MPI communication is given first. The standard mentions two groups of two-sided communication: *point-to-point communication* [For15, chapter 3] and *collective communication* [For15, chapter 5].

Point-to-point communication

In point-to-point communication, there is one sender process and one receiver process, and in the general case, the sender is identified by a call to `MPI_Send`, with a matching receiver calling `MPI_Recv`. Messages from a sender to the receiver are composed of the payload and a so called *message envelope*. As every MPI process belongs to one or more MPI communicator groups, predefined or customized ones, every process carries an identification value – the *rank* – per communicator. Hence, the envelope describes the communicator handle, the rank of both parties under the given communicator, and a tag value.

For a sender, the software developer composes the data to transmit into a local memory location, and calls `MPI_Send` or any of its variants (namely `MPI_{B/R/S}send`) with this memory address and the envelope signature. For a receiver, a call to `MPI_Recv` expects the communicator, a specific source rank or a wildcard, a specific tag or a wildcard, and the address to a sufficiently large local memory buffer to store the incoming payload. The blocking nature of `MPI_Recv` waits for an MPI message that matches the provided envelope signature.

The MPI standard defines multiple send modes to address questions with respect to waiting or buffering of data when a matching receiver call has not (yet) been reached by the other party:

- *Standard mode*: After returning from `MPI_Send`, the provided memory is cleared for future use. If there is no matching `MPI_Recv` at the destination, the implementation decides between synchronizing and buffering.
- *Buffered mode*: After returning from `MPI_Bsend`, the provided memory is either safely buffered by the implementation or transmitted. Buffering is mandatory in the case of not reaching a matching `MPI_Recv` at the destination for not blocking this call. If the local buffer is exceeded, an error status is returned.
- *Synchronous mode*: After returning from `MPI_Ssend`, a matching `MPI_Recv` has been reached at the destination.
- *Ready mode*: A call to `MPI_Rsend` is non-errant only if the destination has already reached a matching `MPI_Recv` call.

MPI demands of an implementation to satisfy the following semantics [For15, section 3.5]:

- *Order*: Messages must not overtake each other.
- *Progress*: For a matching `MPI_Send/Recv` pair in two processes, at least one part eventually ceases blocking for doing progress.
- *Fairness*: No guarantees are given with respect to fairness and process starvation.
- *Extensible buffer space*: For buffered sending, the user must be able to extend the buffer space used by the implementation.

Collective Communication

To spare software developers from doing burdensome work managing point-to-point communication in arbitrarily large communicators efficiently, MPI defines a set of collective operations for fencing processes, and for distributing and reducing data across a communicator.

The nature of collective communication drops the need for tag values at the call level as well as explicit rank specifications in some cases. Designing a communicator setup and its groups can be considered a topic on its own [For15, chapter 6]. For every collective call but `MPI_Barrier`, MPI gives no guarantee for synchronization with other processes of the communicator. A return from any such call only indicates readiness to use or reuse the data at the given memory locations.

As the MPI standard document refrains from imposing architectural details on an implementation, terms like *posting* being used inside of advice sections can be considered as practical implications of designing a communication protocol satisfying the requirements of MPI. One such example demonstrating the suggestion of best-practices in two-sided communication can be found in [For15, section 3.5, p. 40]:

> *synchronous send*: The sender sends a request-to-send message. The receiver stores this request. When a matching receive is posted, the receiver sends back a permission-to-send message, and the sender now sends the message.

Considering a third party managing the communicator's state across an arbitrary number of processes, then a call sequence for a synchronizing `MPI_Send` command on a delayed receiver can be derived as shown by the sequence diagram of figure 2.1. The amount of synchronization exchange on a per-call basis and the additional latency leads to the idea of one-sided communication.

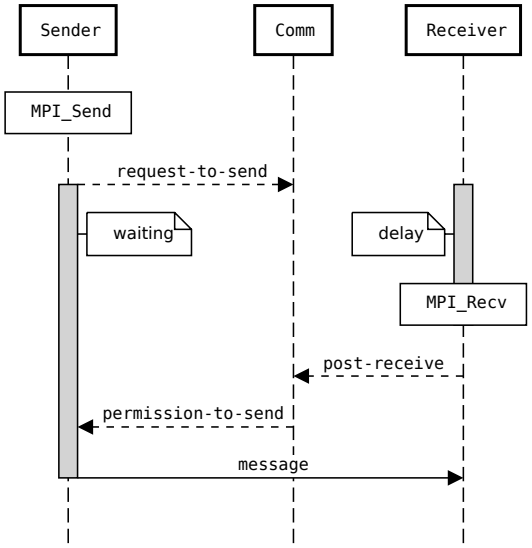


Figure 2.1.: `MPI_Recv` delayed in the synchronous mode.

2.3. One-sided communication

One-sided communication [For15, chapter 11] – or *MPI Remote Memory Access* (MPI RMA) – is supposed to code-wise disengage the receiver from performing a handshake for individual communication actions caused by a sender as demonstrated by figure 2.1. Instead, processes request one or multiple window objects per communicator, with those that serve as a receiver having a range of memory attached. Windows can be classified by their *memory flavor*, depending on their creating call. With all setup calls being collective, these are the standardized flavors:

- *Static, preallocated*, by using `MPI_Win_create` [For15, section 11.2.1]: The user is expected to allocate a suitable amount of memory in advance, letting the MPI implementation know its starting address. A null-value is permitted, disqualifying the caller as a receiver process over this window. The memory size remains fixed during the window's lifetime.
- *Static*, by using `MPI_Win_allocate` [For15, section 11.2.2]: The MPI implementation covers the allocation of the required amount of memory by itself, returning the resulting base address to the user.
- *Dynamic*, by using `MPI_Win_create_dynamic` [For15, section 11.2.4]: Skipping any initial memory allocation, the user dynamically attaches memory to the window using `MPI_Win_attach`, or conversely detaches memory by calling `MPI_Win_detach`.
- *Shared*, by using `MPI_Win_allocate_shared` [For15, section 11.2.3]: A setup that maps the window memory of each process into the local memory address space of every other process. By using `MPI_Win_shared_query`, the user obtains the address of the locally mapped window memory of a given process rank.

The choice of an actual flavor depends on the user's requirements: For use cases that may demand a growing window memory at some point in the lifetime, a user chooses the dynamic over a static flavor. Either way, MPI allows an implementation to choose from two different memory models for publishing remotely accessible window memory: the *unified* model and the *separate* model [For15, section 11.4].

In the unified memory model, the exposed window memory of a process is the same as the locally accessible one. In the separate memory model, this is not the case: The owning process locally accesses only a local copy, expecting a call to `MPI_Win_sync` to synchronize

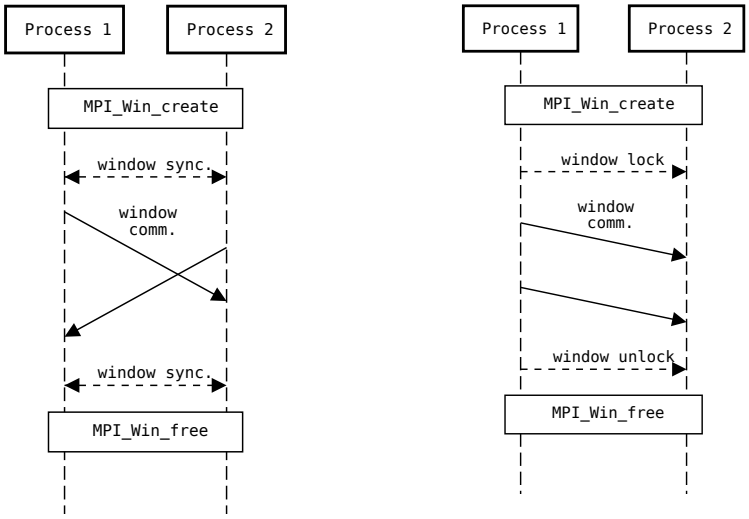


Figure 2.2.: MPI RMA target synchronization: active (left), passive (right).

both the local and public copy. By performing a call to `MPI_Win_get_attr`, asking for the value of the `MPI_WIN_MODEL` attribute of a given window, the user can write portable programs by taking either mode into account, namely enforcing explicit synchronization of the two memory locations. The reason for having two separate memory models is justified not only by backward compatibility but also by allowing implementations to better exploit certain hardware constraints, such as a significant difference with respect to performance of either location.

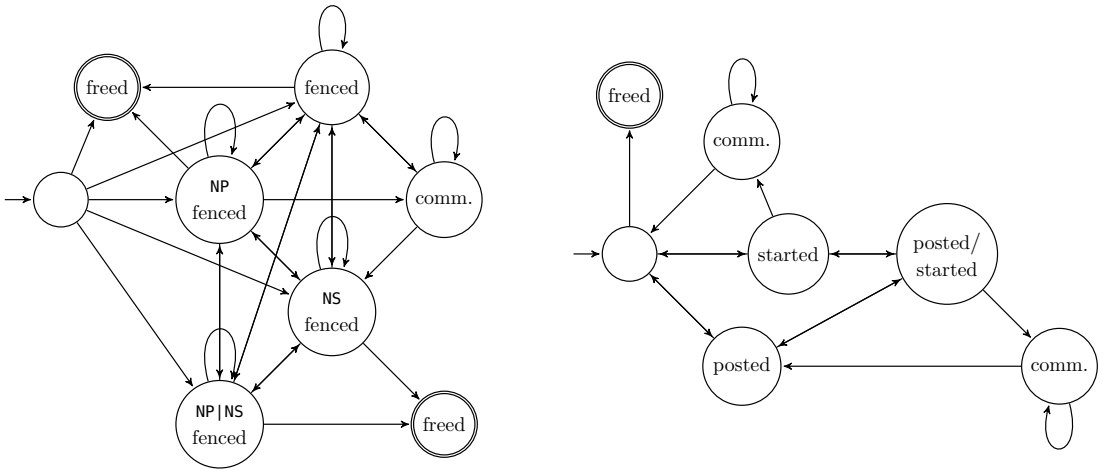
In MPI RMA, communication *epochs* between processes are synchronized by two different modes, *active* and *passive target synchronization* [For15, section 11.5], with the active synchronization providing a simplified *fencing* mechanism and a more flexible one called *general active target synchronization* (GATS). A model of the corresponding message exchange is given by figure 2.2: In the active mode, every involved party actively takes part in synchronization calls but not necessarily in communication calls, whereas in the passive mode, a target shows no active participation in any request at all.

In the following sections, aspects of MPI RMA communication and synchronization are presented in detail, as well as the individual target synchronization modes.

2.3.1. Communication

For the decoupling of communication and synchronization of one-sided MPI communication, when referring to *communication* in the following of the thesis, the given set of communication types is generally addressed:

- *Explicit point-to-point comm.*: This set consists of `MPI_Get` and `MPI_Put`, connecting a single sender to a single receiver [For15, section 11.3.1, 11.3.2].
- *Implicit point-to-point comm.*: When using a *shared*-flavor window and obtaining a local memory mapping for a remote process, any access to that memory region is considered communication.
- *Accumulate comm.*: When using explicit communication, any local operation on received



(a) Fence comm.-sync. FSM with flags `MPI_MODE_NOPRECEDE` (`NP`) and `MPI_MODE_NOSUCCEED` (`NS`).

(b) GATS comm.-sync. FSM.

Figure 2.3.

or submitted data requires synchronization in advance. Primitive tasks, e. g. calculating a sum over all input values at some target process, then require intermediate input buffering and synchronization before the clearance to perform this operation. Accumulate methods help saving this overhead by combining transfer of data and doing an in-place, possibly atomic operation at the desired site. Such methods include, without limitation: `MPI_Accumulate`, `MPI_Fetch_and_op` or `MPI_Compare_and_swap` [For15, section 11.3.4].

As far as applicable, communication also refers to any request-based derivatives of the mentioned methods, such as `MPI_Rget`. It is crucial to note that request-based communication is only legal for accessing processes in the passive target synchronization mode [For15, section 11.3.5].

2.3.2. Synchronization

Calls to synchronization methods define the limits of sequential *epochs*. For any synchronization mode, the given MPI window is stateful and thus its modes can be modeled by a *finite-state machine* (FSM) or a *pushdown automaton* (PDA) with a limited set of legal transitions between states, for example *holding a lock* or *awaiting communication or synchronization*, with the respective set of MPI RMA methods acting as the input alphabet.

Synchronization: Fences

The simplest but also broadest approach to one-sided synchronization is known as *fencing* [For15, section 11.5.1]. Communication must be preceded and synchronized by a call to `MPI_Win_fence`, and *every* process attached to the given window must reach the same call belonging to *this* window in its local execution path before the process may continue. In addition to the generic use of `MPI_Win_fence`, one or more flags may be passed as an argument to this call. Among implementation-specific flags, the MPI standard defines two flags in particular that impose further restrictions:

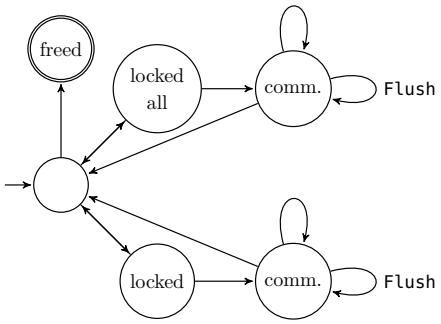


Figure 2.4.: The RMA PTS FSM.

- `MPI_MODE_NOPRECEDE`: This flag indicates that the previous epoch, if given at all, has not issued any communication calls that await synchronization.
- `MPI_MODE_NOSUCCEED`: With this flag being set, the following epoch, if given at all, will not issue any communication.

These flags may be combined over a bitwise `OR`-operation, and these flags must be set equivalently at *every* call to `MPI_Win_fence` that terminates the same epoch at each process [For15, section 11.5.5]. The resulting state diagram from the perspective of a single process is shown by figure 2.3a.

Synchronization: GATS

In contrast to fences, with respect to the set of processes involved, GATS allows a higher granularity, making use of MPI groups [For15, section 6.2.1]. Under a given window, a process either initializes an *access epoch* (via `MPI_Win_start`) that may issue communication to any of the targets of the specified group, or it may also enter an *exposure epoch* (via `MPI_Win_post`) that simply serves as a target for any process of the provided group, or act both as an accessing and a target process [For15, section 11.5.2]. To finish either epoch, processes have to invoke synchronization calls. The local perspective of a state diagram is demonstrated by figure 2.3b.

Synchronization: Passive Target Synchronization

In MPI RMA *passive target synchronization* (PTS), processes either enter the access epoch by using a call to `MPI_Win_lock`, or no epoch at all, serving as a *passive target* without any actions beyond creating and cleaning up the window object [For15, section 11.5.3]. For the time of its access epoch, the accessing process is supposed to acquire locks – shared or exclusive ones – to one or multiple target processes, preceding any communication to them.

While holding a lock, the accessing process is entitled to call `MPI_Win_flush` or any of its variants to wait for any outstanding communication request to finish locally, or both locally and at any specified target process [For15, section 11.5.4]. The state diagram – split into the use of `MPI_Win_lock` and `MPI_Win_lock_all` – is given by figure 2.4.

2.3.3. Correctness

From a more formal perspective, [HDT+15, section 3.1] specifies additional correctness criteria for using MPI RMA beyond the depictions of figure 2.3a, figure 2.3b (p. 13) or figure 2.4 (p. 14). In a shortened manner, these criteria include:

- *Locks*: Every process acquiring a lock must eventually release this lock, and every release of a lock is following the acquisition of this lock. A process must not attempt to acquire a lock to a target it already holds.
- *Fences*: For two or more processes being synchronized over a fence, each process must reach the fence.
- *Exposure*: A window being actively exposed must not be locked at the same time.
- *Passive mode*: PTS methods, namely `MPI_Win_flush` and its siblings, must be called only after acquiring the respective lock, with no fence between.
- *Access*: For every case, the process issuing communication must have entered an access epoch. If the most recent synchronization was a call to `MPI_Win_fence` – without the `MPI_MODE_NOSUCCEED` flag being set –, then the window has entered an access epoch.

Actions

This thesis adopts the concept of *virtual actions* by [HDT+15]. A virtual action has no code-wise representation but merely describes the local effect at a target process, induced by some remote action of another process targeting the respective one, and solely performed by internal mechanisms of the MPI implementation. Under the unified memory model, such an action may then copy locally visible data, or cause modifications to become locally visible in an instant. Under the separate memory model, the effect becomes visible in the public window memory only, and thus for any other process but the target until locally synchronizing the copies.

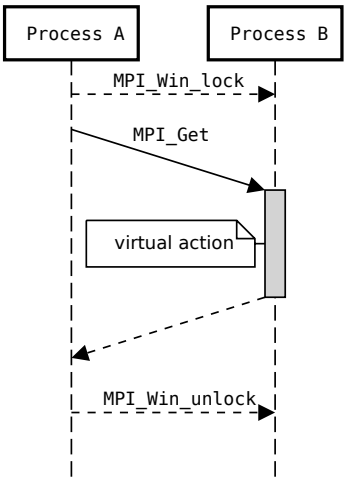


Figure 2.5.: A virtual action.

In the example given by figure 2.5, process `A` entered the access epoch by locking process `B`, a purely passive participant for the time of being locked. The call to `MPI_Get` by `A` towards `B` requests data from the memory exposed by `B`'s window. The MPI implementation dispatches the virtual action transparently at `B` and copies data from the requested frame into the returning submission buffer, finally reaching `A`.

Moreover, the following references to MPI RMA action types will be made in this thesis, not including any kind of synchronization:

- *Local actions*: Local read or write operations on memory that is exposed by a window.
- *Remote actions*: Communication calls, or operations on locally-mapped remote window memory (shared flavor), from the perspective of the invoking process, with a virtual action to be dispatched at the target.

- *Buffer actions*: Remote or local actions on memory that serves as a buffer for a preceding remote action.

Ordering

Given any two actions a_1 and a_2, if a_1 is guaranteed to always have returned before a_2 starts, then a_1 is in *order* before a_2. In a single-threaded process, ordering of a_1 before a_2 is given if a_1 is scheduled before a_2. For multi-threaded or multi-process environments, ordering two actions requires additional measurements for enforcement. For example, in MPI, `MPI_Barrier` provides a communicator-wide method to enforce ordering actions before and after that call [For15, section 5.3].

Consistency

Given any two actions a_1 and a_2, with a_1 ordered before a_2, *consistency* is given if a_2 is guaranteed to always observe the memory effects triggered by a_1. *Local consistency* refers to intra-process consistency whereas *remote consistency* refers to actions taking place at two or more different processes targeting effects on the same memory location. Inspecting the set of synchronization calls, these calls provide guarantees to consistency at different scopes.

Synchronization

When using MPI RMA, ordering actions does not imply consistency between two ordered actions. A programmer not familiar with MPI RMA to the necessary degree may write code as given by the left snippet of listing 2.1. Despite ordering the call to `MPI_Get` before using the returned `data` – possibly a non-zero value by guarantee – the condition may or may not be met at the time of evaluation. Without a call to the applicable synchronization method – as shown by the code snippet to the right – such code will not ensure local consistency.

```
1  uint64_t data = 0;
2
3  MPI_Get(&data, ..., win);
4
5  if (data > 0) {
6    // ...
7  }
```

```
1  uint64_t data = 0;
2
3  MPI_Get(&data, ..., win);
4  MPI_Win_fence(..., win);
5
6  if (data > 0) {
7    // ...
8  }
```

Listing 2.1: Non-synchronized ordering (left), fence-synchronized ordering (right).

By ordering action a_1 before action a_2, and given an appropriate synchronization call s, with a_1 being ordered before s, and s before a_2, a_1 and a_2 are in *synchronized order*. To further examine the meaning of *appropriate*, an overview of MPI RMA synchronization-related methods and methods with respective behavior in that context is provided by table 2.1. The column *Ordering* describes any non-local ordering effects, assuming an otherwise correct use; the column *Consistency* names consistency enactments when returning from the call. For this thesis, *synchronized actions* refers to two actions that fall into different, non-concurrent epochs, or two actions having a suitable `MPI_Win_flush` call ordered between.

Call	Mode	Ordering	Consistency
`MPI_Barrier`	generic	communicator-wide	-
`MPI_Win_free`	generic	window-wide	-
`MPI_Win_fence`	fencing	window-wide	window-wide
`MPI_Win_complete`	GATS	-	local
`MPI_Win_post`	GATS	-	-
`MPI_Win_start`	GATS	by lazily waiting for a matching `MPI_Win_post`	-
`MPI_Win_wait`	GATS	by waiting for a group-wide `MPI_Win_complete`	local
request-based comm.	PTS	-	local (after a `wait`)
`MPI_Win_flush`	PTS	-	selectively local and remote
`MPI_Win_flush_all`	PTS	-	local, remote
`MPI_Win_flush_local`	PTS	-	selectively local
`MPI_Win_flush_local_all`	PTS	-	local
`MPI_Win_lock`	PTS	by waiting for release of an exclusive lock	-
`MPI_Win_lock_all`	PTS	by waiting for releases of all exclusive locks	-
`MPI_Win_sync`	PTS	-	local memory copies (separate memory model)
`MPI_Win_unlock`	PTS	-	local, remote
`MPI_Win_unlock_all`	PTS	-	local, remote

Table 2.1.: MPI RMA synchronization methods.

Conflicts

[DBB+16] provides an overview of conflicting and non-conflicting actions with respect to the targeted window memory region and the memory model. In general, two non-synchronized actions are *in conflict* if

- under either memory model, at least one is non-local and at least one is writing, without both being accumulate actions, *and* their memory operation ranges overlap or
- under the separate memory model, both perform a write and exactly one is local and the other one virtual.

By this thesis, the conflict definition is extended to non-synchronized buffer actions: A remote action eventually writes into a local buffer, and the buffer action attempts to read this memory (shown by listing 2.1, p. 16), or inversely, the remote action eventually reads the buffer for transmission and a buffer action writes to this location.

The natural consequence of any such conflict is a *data race*.

2.4. Static analysis of MPI RMA

In this chapter, a conceptual presentation of MPI one-sided communication was given. This information yields a set of questions that turn out to be suitable candidates for a static analysis, helping a developer to find violations in application code before execution. These candidates include:

- *Intra-path validation*: Given the different window synchronization modes, is there any path that may lead to an illegal sequence of MPI RMA calls? Given the different window flavors, is there any call to an MPI RMA method that is not applicable to that flavor?
- *Inter-path validation*: Is there any path that may have a mismatching number or order of barriers and fences, and thus may result in a deadlock? Is there a mismatch of flags that are expected to be set to the same value for any path and a given epoch of the window?
- *Inter-rank validation*: Considering the use of `MPI_Comm_rank` and branching code sections over rank conditions, is there code without a counterpart with respect to certain window modes?
- *Conflict analysis*: Are there any obvious cases leading to conflicts, notably conflicts in the context of buffer actions or the separate memory model?

3. Static analysis and LLVM

Inspecting MPI communication in the previous chapter, this chapter is dedicated to the topics of static analysis, the LLVM environment, a tool named Phasar and its use of the IFDS problem solution algorithm, as a basic understanding of each topic is required to comprehend how this thesis performs a static analysis on MPI RMA program code.

Software analysis roughly falls into two categories: *static* and *dynamic analysis*. The distinction is made by the requirement to have the software being executed (dynamic) or not (static). Nonetheless, analysis software may combine characteristics of both worlds: One such example is *Valgrind*, working as a dynamic analysis framework by combining an on-the-fly, section-wise disassembly for a static analysis and instrumentation at the run time of the program [NS07].

Neither analysis mode may overcome the other one: A dynamic analysis is limited to actually executed code paths, thus depending on the program's input. Therefore, dynamic analysis serves the need to acquire run time information such as monitoring resource allocation, finding hot code paths and profiling the lifetime for a performance analysis. Static analysis does not suffer from being limited by actual inputs. Much more, given a query of properties to check, without the necessary degree of context taken into account, static analysis easily ends up working on infeasible code paths, reporting false-positives.

In the following section, aspects of static analysis are covered more thoroughly.

3.1. Static analysis

A very fundamental model of any code under analysis is the *control flow graph* (CFG). As a directed graph, a node represents a list of instructions, and an edge connects that node to any successor node. A node is said to *fork* or to *conditionally branch* if it has more than one outgoing edge. A *merge node* describes a node having more than one incoming edge. In general, for this thesis, any *intra-procedural* CFG, denoted as $G = (V, E, s)$, is expected to meet the following criteria:

1. There is one *entry point* node $s \in V$, and only one.
2. Every node $v \in V$ has at most two outgoing edges.
3. The CFG *terminates*: For all nodes $T \subseteq V$ that have no successor, every walk starts at s and ends at some node $t \in T$, or there is no walk for $E = \{\}$.

From a practical view, a software developer is expected to carefully use `goto` or `jmp` instructions as they allow putting a CFG into an infinite loop, violating the termination property. Furthermore, constructing the CFG may terminate prematurely if the target of an edge cannot

be determined statically.

Considering a node n, forking over a condition c into a *true-case*-successor node m_c and into a *false-case*-successor $m_{!c}$, for each state hitting this fork in a graph traversal, a *path-sensitive* analysis will now maintain two new states: One for which c holds, and one for which it does not. In later chapters of this thesis, path sensitivity is shown to be handy but also costly. In a worst-case scenario, the state space will grow exponentially to the number of conditional branches, and grow towards infinity on the occurrence of loops [DLS02].

A static analysis performs a *symbolic execution* of the code if it does not rely on concrete inputs but works on symbols possibly representing a whole class of inputs [Kin76]. The approach of working on concrete inputs is described as *concrete execution*, with solutions being known to combine both approaches, named *concolic* by [BCD+18]. Also, it may serve an analysis purpose to perform a *backward execution* [CFS09], by traversing the instruction list of a node upwards, and over inverted edges, with its counterpart simply named *forward execution*.

3.2. Model checking

Among a rich set of tools for generic and specific tasks[1], one application of static analysis is *model checking*.

For this thesis, the terminology in the context of model checking is taken from [JM09]: Model checking describes the systematic approach to prove properties of software. A *property* is a predicate expected to be fulfilled *conditionally* (*locally*) or *unconditionally* (*globally*). A *program* consists of one or multiple *execution paths*. *Verification* of a property is reached upon the proof that every feasible execution fulfills that property, with an individual execution *satisfying* a property. The *safety property* globally expects that no *error* is ever reached, otherwise the *trace* defines the path leading to the error. The *liveness* property assumes that, eventually, every execution satisfies a given property. *Soundness* of a verification procedure is given if upon indication of safety of a program, the program can be proved for safety. If for every provably safe program the algorithm indicates safety, then the algorithm satisfies *completeness*. In a verification, if the safety property is satisfied by every execution, a program is said to be *correct*. The complement of verification is *falsification*: The mere absence of any safety violation does not imply correctness of the program.

Technically, properties may be expressed in different ways: At the design level of software, the *Object Constraint Language* extends UML to encode properties such as invariants, pre- and post-conditions [BBL10]. At the source code level, depending on the language, developers can specify assertions, such as `assert`[2] in C code, with their untruthful evaluation implicitly specifying an error. For formal systems, *Hoare triples* define a precondition-action-postcondition notation of properties [Hoa69].

With a model as simple as a *finite-state machine* (FSM), defined by an input alphabet Σ, a state set S and its initial state s_0, $F \subseteq S$ being final states, and a transition function $\delta : \Sigma \times S \to S$, an equivalently simple property can be tested for satisfaction by reaching a final

[1] cf. Wikipedia – `https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis`, last seen 2018-12-03.

[2] `assert` of the C library primarily acts as a run time assertion.

state, with detecting violations by observing attempts to perform transitions not specified by δ.

Model checking tool techniques include automata, such as for Blast [BHJM07], LTL or CTL$^{(*)}$ specifications, for example in LTSMin [KLM+15], but also Prolog-based systems exist [FOSM17].

3.3. Considerations & Motivation

The goal of this thesis is to perform the verification of programs in the context of the MPI one-sided communication API using static analysis for questions that have arisen in section 2.4. Therefore, before advancing to the next section, it is worth examining the environment of MPI for technical aspects with respect to static analysis, and to make expectations explicit to potentially address a wide audience of developers.

For programming languages, first-class citizens of the MPI specification are C and Fortran. By addressing C, its taller sibling C++ is following closely, and possibly other, header-compatible languages as well[3]. Taking C, C++ and Fortran into account, syntax-wise, every language has undergone an evolution over time, with future specifications to come[4]. Languages that enjoy intermediate bindings to MPI are not primarily focused by this thesis. Preferably, when building a solution to perform verification of MPI RMA properties by a static analysis,

1. it does not depend on code-invasive preparations,
2. it works on C, C++ and Fortran program code,
3. for each language, it works on every non-archaic specification,
4. it should not address any language instance itself but work on a normalized, cross-language representation,
5. and this representation, by default, does not depend on any hardware platform and
6. it can still link analysis results of that representation to single lines of the original source code.

When considering a representation that is independent of any programming language, it is worth inspecting the targets that compilers emit code for: CPU-specific or CPU-neutral.

A CPU-specific binary representation violates property 5, as it is desirable not to work against platform-specific properties as well as vendor-specific extensions, and also for the reason of decreased portability when cross-compiling is not an option. CPU-neutral representations include formats such as Java bytecode or the *Common Language Infrastructure*, formally known as ECMA-335. Unfortunately, the language support in either environment for properties 2 and 3 turned out to be poor.

After taking various alternatives into consideration, the choice for the LLVM environment was made as it promises to seamlessly fulfill all of the desired properties.

[3] examples include: Objective-C and Objective-C++
[4] See ISO/IEC doc. no. N4778 for C++20.

3.4. LLVM

The *Low Level Virtual Machine* (LLVM) environment comprises a set of compilation-related tools, specifications and development libraries that aim to connect various programming languages to arbitrary execution platforms, providing a middleware for toolchain developers to work with.

LLVM dates back to 2002 as a graduation project by C. Lattner [Lat02], with its permissive license attracting a wide range of commercial contributors such as *Apple Inc.* or the *NVIDIA Corporation*[5]. As of today, in the open-source compiler community, together with the *GNU Compiler Collection* (GCC), the C and C++ compilers by LLVM have become major players [s.r18].

3.4.1. Architecture

To understand the benefits of using LLVM in this thesis, a short description of the architecture of LLVM is given first.

There are three major stages when compiling source code in the classical sense, also shown by figure 3.1:

1. A *front end*: For a given programming language, a front end compiler takes care of language-specific tasks such as preprocessing, tokenization, constructing an *abstract syntax tree* (AST) and emitting a new representation. Instead of being coupled to a specific, maybe proprietary representation, the emitted code is processable by the LLVM middleware.
2. The LLVM *middleware*: LLVM specifies an *intermediate representation* (IR)[6], a RISC-like assembly language with *static single assignment* (SSA) characteristics. Software in the IR stage can be widely processed for different tasks.
3. A *back end*: Out of the IR, the back end component emits a target-specific representation. This includes CPU platforms, but also soft targets have been shown to be feasible [Zak11].

LLVM is known for its C/C++ compiler named *Clang*. For Fortran, there is *Flang*[7], maintained by NVIDIA and not directly associated to the canonical LLVM tool set. From a user's perspective, it is not required to manually go through these stages, as the compiler bundles provide *driver* programs that assist in maintaining a classical compile-and-link build setup.

For its interesting properties and the fitting of a static analysis, the middleware deserves more attention. In general, the IR is encoded by two competing formats: a binary or *bitcode* representation (`.bc` files) and a human-readable representation (`.ll` files). Both formats encode the same information and can be used interchangeably by most of the LLVM tools and the library. While the bitcode representation benefits from a much better degree of compression, when referring to the IR in this thesis, the human-readable format is used for any demonstration.

The LLVM tool suite contains a couple of interesting programs that are presented briefly:

[5] LLVM Users – `https://llvm.org/Users.html`, last seen 2018-12-04.
[6] LLVM Language Reference Manual – `https://llvm.org/docs/LangRef.html`, last seen 2018-12-04.
[7] Flang on GitHub: `https://github.com/flang-compiler/flang`, last seen 2018-12-04.

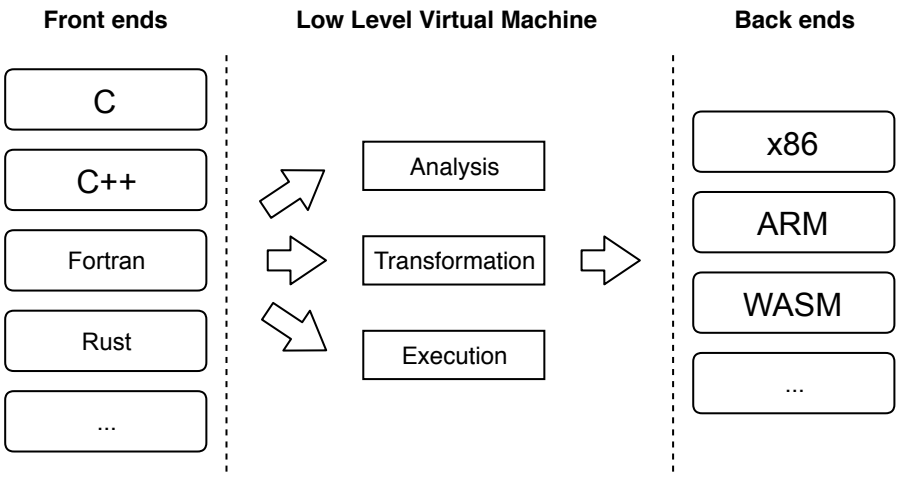


Figure 3.1.: Conceptual architecture of LLVM.

- `llc`. The compiler for translating IR files into the representation of the requested back end.
- `lli`. Any IR file can be executed, and `lli` performs this task by either directly interpreting the IR file, or by doing a just-in-time compilation. As linking to external libraries has not taken place at this stage, such libraries must be loaded explicitly for this tool to run a respective IR file.
- `llvm-link`. On the intent to pack multiple IR files together – as they result from translating individual source files –, `llvm-link` acts as the linker. This way, multiple IR module files can be melted into single one, allowing any IR analysis to work on a wider scope or even the complete program at the IR level.
- `opt`. The `opt`-tool allows a wide range of analysis, transformation and optimization operations on a given IR file. An optimization pass may be predefined (for example by `-Ox` flags) or selected more granularly. A tool developer can also choose to create a plug-in for `opt` for both analysis and transformation of the IR[8]. Choosing to write a plug-in of `opt` over a stand-alone tool enables the user to better combine the use of `opt` – or its driver – with other available options, likely making the toolchain more efficient.

Let the following example be given, depicted by figure 3.2: A C source code base consists of two files, `main.c` and `mod.c`. While `main.c` contains the `main` entry point, plenty of its logic is located in `mod.c`. In the most basic attempt to create an executable program `exec`, the user invokes the driver of the compiler that takes care of compiling, assembling and linking. For LLVM and Clang, the `clang` program acts as the mentioned driver. With LLVM, the toolchain can be decomposed comfortably: With providing the appropriate flags, the C files will be translated to individual IR files. If desired, the two components of the example can be linked at the LLVM stage already (`program.ll`), allowing the user to make use of any LLVM-based tool for either scope of the program composition.

In the context of a verification tool, this renders two handy features: Considering a large code base of hundreds of files, analyzing the IR of a single code module file possibly fails to

[8]Writing an LLVM pass – `https://llvm.org/docs/WritingAnLLVMPass.html`, last seen 2018-12-05.

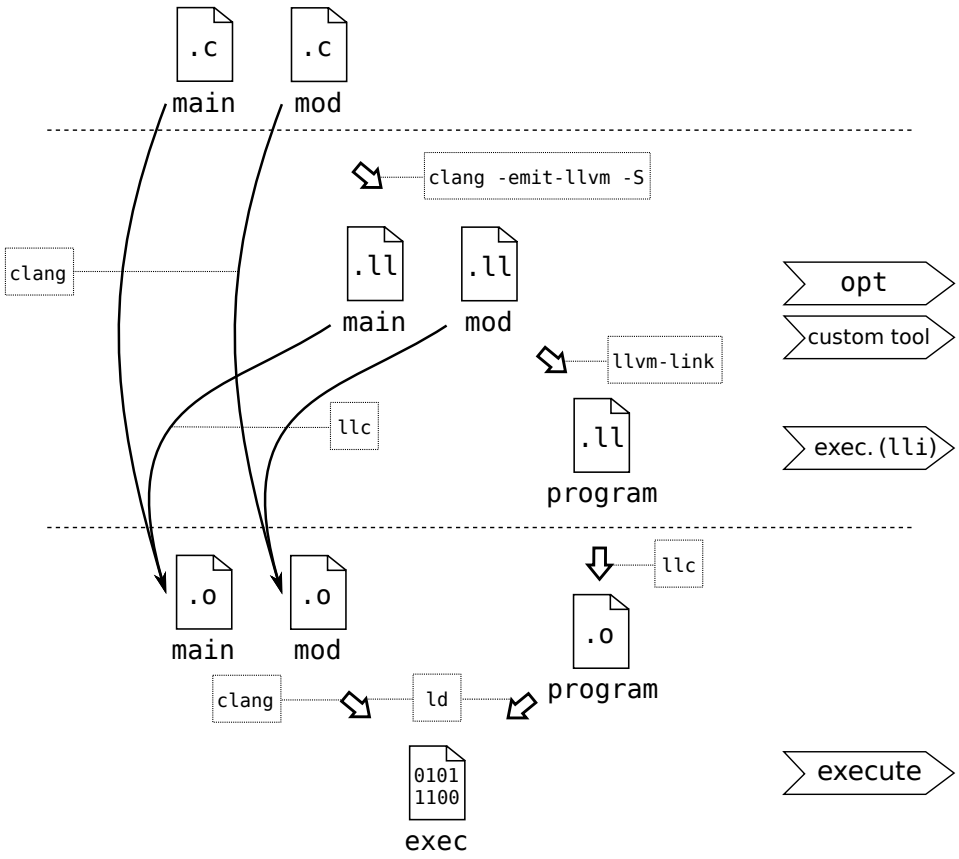


Figure 3.2.: An LLVM toolchain example.

fulfill the completeness of the analysis. On the other hand, the less the program is bloated by unrelated code sections the less time is to be consumed by the tool. LLVM allows the user to easily compose the unit under analysis, balancing these aspects within the limits of the program's structure.

3.4.2. Anatomy of the IR

For a basic understanding on how to work with the IR, a short introduction is given by the example of listing 3.2 that has been generated from the C++ code shown by listing 3.1, ignoring any MPI correctness for the sake of demonstration.

An IR file unit is called a *module* that contains multiple sections, shortly described in a line-wise manner by dissecting listing 3.2 (p. 26):

- 1, 2: The module has been created from a C++ file named `demo.cpp`.
- 3, 4: Derived from settings or from an automatic host detection, the front end compiler specified module attributes regarding the target architecture and memory layout.
- 6: This defines a *module variable* (also *global variable*) named `@window` of type `i32`, a signed 32-bit integer.

```
1  #include <mpi.h>
2
3  MPI_Win window;
4
5  int main (int argc, char **argv) {
6    int assert = 0;
7
8    if (argc > 0) {
9      assert = MPI_MODE_NOSUCCEED;
10   }
11
12   MPI_Win_fence(assert, window);
13
14   return 0;
15 }
```

Listing 3.1: The example source code for listing 3.2.

- $8 - 27$: A definition of a *function* named `@main`, having two parameters (`i32` and `i8**`, a pointer to a pointer to `i8`) and returning a value of type `i32`.
- 8: Inside a comment, the compiler has given some hints on attributes of the function, such as `norecurse` for indicating the absence of recursion.
- $10, 12$: Here, a *local variable* of type `i32` and name or *symbol* `%3` has been declared, getting its initial value set to `0`; more precisely, `store` requests `0` to be written at the address of `%3`.
- $14 - 16$: The value of `%4` is loaded into `%7`, followed by a signed-greater-than comparison (`icmp sgt`) against `0`. Its Boolean result (in `%8`, type `i1`) is then used in a *conditional branch*. In the true-case, the code is supposed to continue at what follows after `<label>:9`, otherwise after `<label>:10`.
- $10 - 16, 18 - 20, 22 - 26$: Each of that block of lines describes a *basic block* (BB). A BB possibly has predecessors (hinted by a `preds`-comment) and ends on a *terminator* instruction such as a branch instruction or the return from the function.
- 19: The constant `16384` is possibly an implementation-specific value for `MPI_MODE_NOSUCCEED`.
- 20: This is an *unconditional branch* instruction, jumping to `<label>:10`.
- 25: A *call* to the declared method `@MPI_Win_fence`, providing two arguments (`%11`, `%12`), and its returned value described by `%13`. The whole line is a *call site*.
- 26: The function always returns `0`. With respect to the call site in line 25, this instruction is its corresponding *return site*.
- 29: The call in line 25 leads to a function that is not defined by the given module. Instead, solely its *declaration* is made public, thus awaiting linking or dynamic loading at a later stage.
- $31 - 34$: If the compiler was run with the request to include *debugging symbols*, the symbols are found amended to the code. For example, in line 19, the *metadata node* `!dbg !109` links the instruction to metadata in line 34. It contains the information that the originating line in the source code file was some line 9 of scope `!110` (function `main`) that belongs to file `!3` (`demo.cpp`).

```
; ModuleID = 'demo.cpp'
source_filename = "demo.cpp"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@window = global i32 0, align 4, !dbg !0

; Function Attrs: noinline norecurse optnone uwtable
define i32 @main(i32, i8**) #0 !dbg !93 {
  %3 = alloca i32, align 4
  ; [...]
  store i32 0, i32* %3, align 4
  ; [...]
  %7 = load i32, i32* %4, align 4, !dbg !105
  %8 = icmp sgt i32 %7, 0, !dbg !107
  br i1 %8, label %9, label %10, !dbg !108

; <label>:9:                                      ; preds = %2
  store i32 16384, i32* %6, align 4, !dbg !109
  br label %10, !dbg !111

; <label>:10:                                     ; preds = %9, %2
  %11 = load i32, i32* %6, align 4, !dbg !112
  %12 = load i32, i32* @window, align 4, !dbg !113
  %13 = call i32 @MPI_Win_fence(i32 %11, i32 %12), !dbg !114
  ret i32 0, !dbg !115
}

declare i32 @MPI_Win_fence(i32, i32) #2

!3 = !DIFile(filename: "demo.cpp", directory: "...")
; [...]
!110 = distinct !DILexicalBlock(scope: !106, file: !3, line: 8, column: 17)
!109 = !DILocation(line: 9, column: 12, scope: !110)
```

Listing 3.2: LLVM IR of the source code of listing 3.1.

The RISC'ish nature of the IR specification can be identified by the requirement to always explicitly fetch values from a location (the `load` instructions), as well as explicitly doing write-backs (the `store` instructions). For the property of SSA, no value identifier must ever be reassigned.

3.4.3. Tool development

LLVM ships a development kit that it is designed to work on the IR programmatically and at full scale. When not using any third-party bindings or the C interface provided by LLVM, its use requires a bit of familiarity with C++.

Essentially, a call to `llvm::parseIRFile` parses any provided IR file and performs an object-oriented layout of that code in memory (the *IR tree*). Thus, its entity-relationship diagram of classes shows its close relation to the file layout. A trimmed diagram, showing essential classes and relations is given by figure 3.3: Out of `llvm::BasicBlock` and `llvm::Instruction`, for some given `llvm::Function`, an intra-procedural CFG can be modeled.

3.5. Phasar

Phasar[9] (also written *PhASAR*) is an example of software developed on top of LLVM. With its first commits to the code repository dating back not earlier than 2017, and currently under active development by the *Paderborn University*, it promotes itself as a static analysis framework [SHB19]. At the time of writing, it requires LLVM v5.0.

It acts both as a framework that enables developers to write plug-ins (similar to `opt`) or – as for this thesis – provides all necessary libraries to be integrated into a custom standalone solution. An integration of a Phasar data-flow problem solution pass into the link phase of LLVM is currently under development[10].

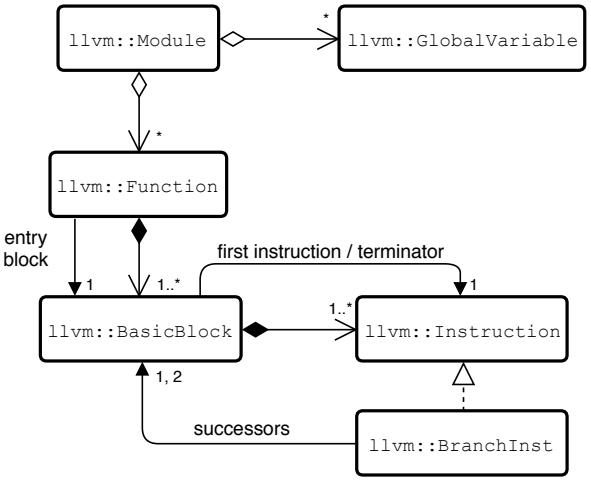


Figure 3.3.: LLVM IR class layout.

Roughly, in order to solve a *data-flow problem*, Phasar performs the following steps:

1. For any given LLVM module, on a per-function basis, it performs a *points-to analysis* for a later attempt to resolve function pointers to their actual targets.
2. It creates a CFG from the module by the desired property: intra-procedural, inter-procedural or backwards inter-procedural.
3. Lastly, it solves the data-flow problem that has been specified by the user over the CFG.

Considering these steps essential for writing certain types of a static analysis, the developer is spared from implementing these tasks manually, permitting to focus on the actual data-flow problem. The next section describes the topic of data-flow problems in detail.

3.6. Data-flow problems

A *general class of data-flow problems* as described by T. Reps et al. consists of a CFG, a finite set D of so-called *facts*, a *distributive operator* (also *meet* or *merge operator*: the *union* or *intersection* operation) and a *data-flow map function* (or simply *flow function*) that maps a *fact state* S_i to another fact state $S_{i'}$ at an edge of the CFG [RHS95]. This fact state is a map of each fact to its liveliness, i. e. $D \to \{\top, \bot\}$. From now on, when referring to the node of a CFG, it is no longer meant to be a *basic block* (BB) but a single instruction of the IR.

By the analysis of a node of the CFG, the flow function may turn a fact alive, as much as it may kill the fact again at a later point in the CFG. A model like this is called a *Gen-Kill problem*. A very simple instance of this problem is the analysis for uninitialized variables.

[9] Phasar.org – A LLVM-based static analysis framework – `https://phasar.org/`, last seen 2018-12-05.
[10] Phasar.org – Using PhASAR as a LLVM full-LTO pass: `https://phasar.org/2018/12/20/using-phasar-as-a-llvm-full-lto-pass/`, last seen 2019-01-20.

Modeling the analysis for uninitialized variables by the definition of a data-flow problem is presented as an example by figure 3.4 on the excerpt of a CFG of a program with variables `a, b`. D represents the set of uninitialized variables, written as $D = \{a^\dagger, b^\dagger\}$, such that liveliness of a fact represents undefinedness. The flow function is denoted by λ. Assuming a previous definition of `b` already, after leaving node n_1, the variable `a` has been shown to be declared without a definition: λ generates $a^\dagger$. Its definition occurs not earlier than node n_3, so by leaving this node, λ kills the fact $a^\dagger$. A static analysis using λ identifies the use of `a` at node n_2 over its undefinedness. The property specified by *on use, every variable must be defined* is violated.

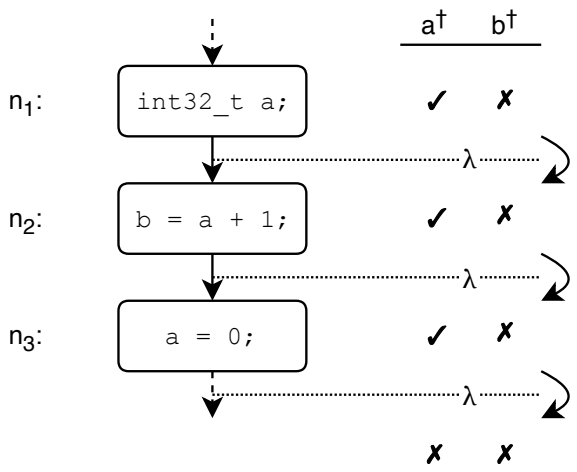


Figure 3.4.: An analysis for uninitialized variables.

For the outgoing edges of every merge node, two or more fact states need to be consumed. To determine the fact state S_i after a merge node, the meet operator $\bigotimes \in \{\sqcup, \sqcap\}$ is applied over all incoming fact states ($S_i = S_I \bigotimes S_J \bigotimes \ldots$). Depending on the problem, the applicable operator needs to be determined. If the fact state of the problem can be modeled over a vector of bits, such a problem belongs to the class of *bit-vector problems*, with $\sqcup$ representing the bitwise disjunction and $\sqcap$ the bitwise conjunction. In its more general form using sets: $\sqcup \cong \cup$, $\sqcap \cong \cap$.

Modifying the previous example for containing a branch with the definition of `a` taking place in one, and only one branch (going over node n_2 in figure 3.5), using the $\sqcup$ meet operator, the fact $a^\dagger$ will then be still alive after node n_3. This way, the analysis can make use of the fact that the previously specified property is violated when *meeting over all paths*.

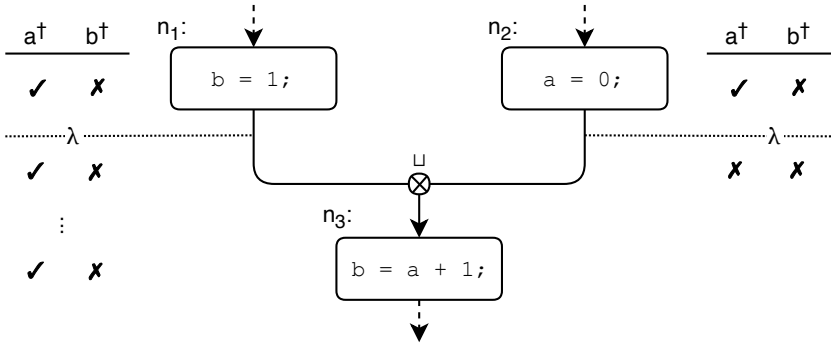


Figure 3.5.: Uninitialized variables in a meet-scenario.

For any non-trivial program, there usually is not just a single CFG but a set of CFGs, each one representing a non-inlined procedure. In the environment of inter-procedural CFGs, data-flow problems may be expressed as an IFDS problem, presented by the next subsection.

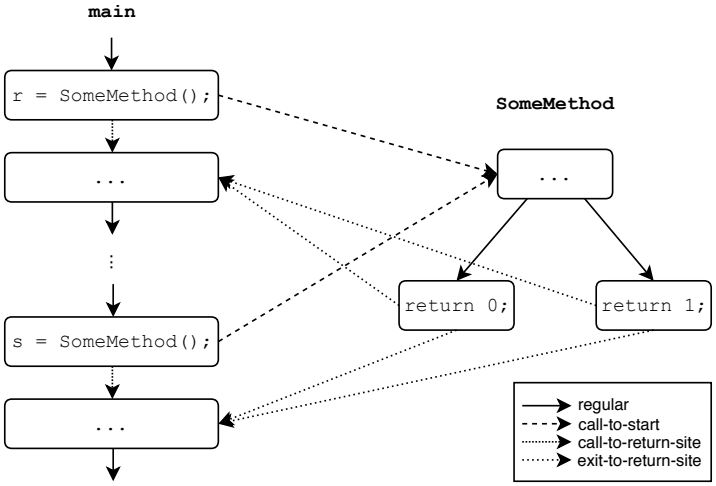


Figure 3.6.: An inter-procedural CFG for IFDS problems.

3.6.1. IFDS problems

T. Reps et al. introduce an algorithm to solve data-flow problems for the class of *inter-procedural, finite, distributive, subset problems* (IFDS problems) [RHS95]. Instead of a single CFG, this class is extended to the super-set of all available CFGs of the program. Given two CFGs $G_a = (V_a, E_a, s_a)$ and $G_b = (V_b, E_b, s_b)$, with $c_a \in V_a$ being a call site of G_a to the procedure modeled by G_b, and $r_a \in V_a$ being a return site of c_a, then the super-set CFG

$$G^* = (V^* = V_a \cup V_b, E^* = E_a \cup E_b \cup E_I, s^*)$$

contains all CFGs, and for each intra-procedural call three additional edge types, found in E_I:

1. A *call-to-start* edge from c_a to s_b.
2. One or more *exit-to-return-site* edges, one for each return node of V_b to r_a. In the publication, a CFG is expected to have one unique exit node. Since the case of having multiple exit nodes can be modeled equivalently by virtually inserting one super-exit node with edges incoming from each exit node, this constraint is dropped here.
3. A distinct *call-to-return-site* edge from c_a to r_a.

In an example visualized by figure 3.6, there is a CFG for a method `main` that carries the program's entry point s^*. There exist two calls to a method named `SomeMethod` that branches into two different exit nodes.

A flow function λ will also be applied to these new types of edges. This way, a property can be tested when *meeting over all valid paths*, i. e. after properly returning from any call.

The proposed algorithm, the *Tabulation Algorithm*, calculates fact liveliness as a graph reachability problem. First, besides the problem-specific facts, D is enriched by a globally live fact: the *zero* or *null fact*, denoted by $\perp$; every flow function preserves this fact. Considering two nodes $\{n_1, n_2\} \subseteq V^*$ with $e = (n_1, n_2) \in E^*$, $\lambda_e : S_{n_1} \mapsto S_{n_2}$, λ_e falls into one of these

types while $\bot \notin B$:

Name	**Definition**
identity	$\lambda S.S$
generator (of facts A)	$\lambda S.S \cup A$
killer (of facts B)	$\lambda S.S - B$
generate A, kill B	$\lambda S.(S - B) \cup A$

The *exploded super-graph*

$$G^{\#} = (V^{\#}, E^{\#})$$
$$V^{\#} = V^* \times D, \text{ the } \textit{node-fact nodes}$$
$$E^{\#} = \{\langle m, d_1\rangle \rightarrow \langle n, d_2\rangle | e = (m, n) \in E^*, \lambda_e \text{ yields } d_2 \text{ for } d_1\}$$

is the graph to be constructed by the Tabulation Algorithm. The very initial state of facts alive is predefined: $S_{s*} = \{\bot\}$. Describing the algorithm prosaically gives a sufficiently good idea of the procedure:

$G^{\#}$ acts as the trace-graph to store information on the liveliness state per fact at each node in $V^{\#}$, with $E^{\#}$ describing the journey of every fact across reachable nodes over its lifetime, but also mapping the generation of new facts (*children*) triggered by that fact (*parent*) over some λ_e. The algorithm terminates if either all facts have been killed (except $\bot$) or reached the end of the CFG. While building $G^{\#}$, choosing the next available fact to be processed by a flow function, the choice where to continue is possibly subject to a scheduling mechanism. For a developer implementing a flow function for some given edge, this has the following practical implications: Given a fact to evaluate in a flow function, one must not rely on any assumption

- regarding the progress of the reachability analysis of the parent, any sibling or any child fact,
- regarding the order of facts completing the reachability analysis,
- regarding the schedule of exploration given conditional branches and
- regarding the preemption of its own processing throughout the graph.

In a worst-case scenario, the number of steps to finish a finite set of facts is $O(|E| \cdot |D|^3)$.

3.6.2. Implementation of IFDS problems in Phasar

In Phasar, the implementation of the Tabulation Algorithm bases on the work of [NLR10] that proposes some modifications to the original algorithm, most notably improvements to the required space when performing the analysis, to the modeling of exit-to-return edges and considerations when working in an SSA environment or on polymorphic facts.

The meet operator is chosen to be $\sqcup$. Although Phasar attempts to implement the algorithm without being tightly coupled to the LLVM IR, this thesis does not make any consideration beyond that.

Any IFDS problem is expected to be expressed as a sub-class of `psr::DefaultIFDSTabulation-Problem`, using a type templatization:

- `N`. The type of the CFG nodes.
- `D`. The type of facts in D.
- `M`. The type of the CFG instances.
- `I`. The type of the CFG, such as a backward or forward CFG.

Thus, for LLVM, the declaration shown by listing 3.3 is the default way to model an IFDS problem: The node-granularity is set to the instruction level, facts are of type `llvm::Value*`[11], and the inter-procedural forward CFG (`psr::LLVMBasedICFG`) consists of CFGs of functions of all modules that are known to Phasar.

```
class IFDSTabulationProblem :
  public psr::DefaultIFDSTabulationProblem<
        const llvm::Instruction*
      , const llvm::Value*
      , const llvm::Function*
      , psr::LLVMBasedICFG&
    >;
```

Listing 3.3: An IFDS problem type declaration for LLVM in Phasar.

In Phasar, a flow function is an object, instantiated by a class with the abstract base class `psr::FlowFunction`, detailed by listing 3.4.

```
template <typename D> class FlowFunction {
public:
  virtual ~FlowFunction() = default;
  virtual std::set<D> computeTargets(D source) = 0;
};
```

Listing 3.4: The base class of a flow function in Phasar.

The developer's task for using Phasar is sketched as follows:

1. Implementing any required, non-trivial flow function. For some fact `source` given by an invocation of `computeTargets`, a set – possibly empty, killing `source` – of facts is returned. A parameterization of the flow function with respect to the CFG is taking place in the next step.
2. Overwriting a limited set of methods imposed by `psr::DefaultIFDSTabulationProblem`, the developer needs to deploy the desired flow function when being requested. Within the predefined methods, context on the type of the edge and the surrounding nodes is provided. Phasar provides some generic flow functions that supersede a manual implementation of generic flow functions, such as `psr::Gen`, `psr::GenIf`, `psr::Kill`, `psr::Identity` and more.
3. Specifying a fact instance that represents $\perp$.

[11] A pointer to the base class of any IR token.

The methods by `psr::DefaultIFDSTabulationProblem` that ask for deploying a flow function are the following ones:

- `getNormalFlowFunction (N a, N b)`. A call on a regular edge between two nodes `a` and `b`.
- `getCallFlowFunction (N c, M m)`. The case of a call-to-start edge from a call-site node `c` to some function `m`.
- `getRetFlowFunction (N c, M m, N e, N r)`. The case of an exit-to-return edge from a call-site node `c`, a return-site node `r` and an exit node `e` of method `m`.
- `getCallToRetFlowFunction (N c, N r, std::set<M> m)`. Calling for a flow function for an edge between call-site `c` and return-site `r`, returning from all function candidates `m`.
- `getSummaryFlowFunction (N c, M m)`. This call precedes `getCallFlowFunction` if there is a call from `c` to a defined function `m`. Otherwise, this is the place to set up a flow function that processes facts over a declared function `m`, such as calls to methods of the C run time library or MPI.
- `initialSeeds ()`. In this call, the developer specifies the initial facts and their place of generation. In the most basic case, there will be one zero fact assigned to the very first instruction of the `main` procedure. Any customization of the initial fact setup must take place here.

This interface design allows taking care of LLVM IR specificities, such as an explicit assignment of a return value to the call-site instruction, and not its return site that is otherwise sufficient to model stack- or register-based returning of values on common hardware architectures.

Multiple calls to `getCallFlowFunction` may arise from the fact that a function pointer is located at the call-site, or by performing a call to a virtual function table. In both cases, Phasar attempts to find suitable candidates. For this purpose, Phasar offers a set of look-up resolvers.

In the next chapter, by making use of Phasar, static analysis of MPI RMA programs is shown to be modeled as an IFDS problem.

4. Static analysis of MPI RMA programs

This chapter describes the idea of how to model the testing of MPI RMA properties as sketched by section 2.4 as an IFDS problem by making use of Phasar. It covers the general modeling of the problem as well as individual property tests that have been enabled by this model.

Part of this thesis is an implementation of a tool to perform a static analysis at MPI RMA code on the LLVM IR stage. This chapter describes the approach to the implementation using Phasar (section 4.1) and the property tests that have been created (section 4.2). Details on the setup of the tool, its precise use and analysis output are presented by appendix A.

4.1. Modeling

To address a wider range of property tests for MPI RMA code, the tool that has been implemented for this thesis consists of two steps: One makes use of an IFDS problem model to explore all, possibly infeasible paths to extract a broad set of MPI RMA operations. In the second step, execution paths undergo further intra- and inter-path property tests.

4.1.1. Process pipeline

For providing a first overview over the whole analysis process, each step – whether required, optional or skippable – is sketched as shown by the abstract CFG of figure 4.1.

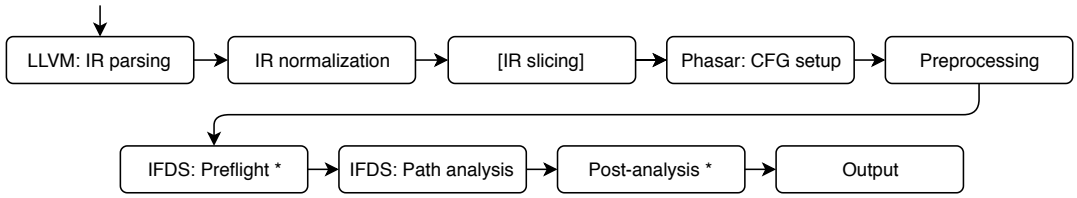


Figure 4.1.: Steps of the static analysis; optional steps in brackets, skippable steps marked by *.

In the order of operation:

1. *IR parsing.* The LLVM library is given a file path to some IR module to set up the object tree of its components.
2. *IR normalization.* In order to make a wide set of legal IR programs work with the following algorithms, the IR is temporarily modified, with more details given by subsection 4.1.2.
3. *IR slicing.* An optional step, resulting from considerations of subsection 5.2.2. If requested, the tool attempts to cut out parts of the IR that are seemingly irrelevant.

4. *CFG setup.* The solver framework of IFDS problems expects a CFG subject to certain properties. In this step, the IR tree is processed, including the points-to analysis to resolve any function pointers, to lay out such a CFG. A manual adjustment on that step is presented by subsection 4.1.2.
5. *Preprocessing.* While the CFG remains untouched, the tool performs an analysis over the IR that is required to work with loops inside of an intra-procedural CFG, discussed in detail by subsection 4.1.5.
6. *Preflight.* A skippable step. A computationally cheap single-fact IFDS problem that acquires statistics on the CFG with respect to the following full-scale analysis. While not essential, it is especially helpful to initially estimate the feasibility of the analysis of the given IR module, and also to measure the impact by the optimization of step 3.
7. *IFDS problem.* The core of the static analysis, finding execution paths for MPI RMA operations, and also doing an on-the-fly model checking. In this thesis, two different IFDS problem models have been implemented.
8. *Post-analysis.* A skippable step. A set of property tests that currently do not fit into the previous step, such as inter-path analyses. For reasons of computational complexity, this step can be skipped.
9. *Output.* In the final step, the program writes user-friendly information into CSV files, reporting any violations that have been identified by the previous two steps.

4.1.2. Preparations

From observations during the development, and also by examining the LLVM IR specification closely, two issues require to be addressed before handling any semantic concerns of the static analysis:

`switch` unrolling

Previously stated in section 3.1, a CFG is expected to consist of nodes with at most two outgoing edges only. This naturally conflicts with the *generalized branch instruction* (`llvm::SwitchInst`) of the LLVM IR, allowing n successor nodes. While this is a handy feature to ergonomically model simple C-like `switch` statements, it asks for more generalized algorithms in various subsequent sections of this thesis.

Instead, in a first step examining every *basic block* (BB) of every function of every module, the last instruction of a BB is tested for being a `llvm::SwitchInst` and then rewritten by a chain of new BBs, connected by dichotomous conditional `llvm::BranchInst` instructions.

This measurement is nowhere handling a rare edge case as Clang has been shown to emit `llvm::SwitchInst` instructions on the occurrence of `switch` statements in simple C code. In listing 4.1, there is an example of such C code and its Clang-native IR translation. The `switch` statement on the right is then replaced by a jump to the equivalently unrolled version listed below, finally leading to the `default` case destination (label `%18`).

The cost of rewriting `switch` instructions grows linear to the total number of the respective destinations inside of the IR module.

```
switch (input) {
  case 1: /*...*/
    break;
  case 2: /*...*/
    break;
  case 3: /*...*/
    break;
  default:
    break;
}
```

```
; native version
switch i32 %8, label %18 [
  i32 1, label %9
  i32 2, label %12
  i32 3, label %15
]

; unrolled version:
%22 = icmp eq i32 %8, 1
br i1 %22, label %9, label %unroll_1

; <label>:unroll_1:
%29 = icmp eq i32 %8, 2
br i1 %29, label %12, label %unroll_2
; ...
```

Listing 4.1: A C `switch`-statement (left) and the IR representations (right).

Flang function pointers

On the first tests of MPI IR code of Fortran programs generated by Flang, Phasar always terminated suspiciously early, leaving large portions of the IR unseen. The reason lies within a certain pattern of invoking declared methods over function pointers, with the difference between Clang and Flang generating the respective IR shown by listing 4.2.

```
%25 = call i32 @MPI_Get(i8* %21, ...)
```

```
%47 = bitcast void (...)* @mpi_get_ to void (...)
call void (i8*, ...) %47(i8* %37, ...)
```

Listing 4.2: A call to `MPI_Get` in Clang (left), Flang (right).

As Phasar otherwise terminates early when unable to identify at least one target, the problem has been solved preliminary by sub-typing the Phasar components `psr::OTFResolver` (a function pointer look-up) and `psr::LLVMBasedICFG` (an inter-procedural CFG layer) to heuristically test for declared procedures cloaked behind `llvm::BitCastInst` casting instructions.

The modification contributes a constant number of operations to the overall costs of the existing procedure of Phasar.

4.1.3. IFDS problem fact domain

In this section, the fact domain D is presented in detail. Technically, the problem's signature for Phasar is comparably simple. Shown by listing 4.3, the only difference to the default type-templatization for LLVM is the use of a pointer to a `Fact` object instead of `llvm::Value`.

For Phasar using the data type `std::set<D>` for managing the facts of the domain, with `D` being a pointer to a problem-surviving object in the heap memory, using `Fact` this way allows a feature-rich, polymorphic data-flow modeling.

Currently, the tool is modeled over the following sub-types of `Fact`: `Null`, `Comm`, `Win`, `WinGhost` and `IntConst`. The modeling was inspired by an IFDS problem model of a *taint analysis* for its API-based fact generation and journey throughout the code. In a taint analysis, certain instructions of program code take inputs by the user, and that input is considered tainted,

```
1    class IFDSTabulationProblem :
2      public psr::DefaultIFDSTabulationProblem<
3            const llvm::Instruction*
4          , Fact*
5          , const llvm::Function*
6          , psr::LLVMBasedICFG&
7        >;
```

Listing 4.3: The customized IFDS problem type declaration.

generating a representing fact. For example, from the perspective of security, it is beneficial to know whether a tainted input reaches a sensitive point without sanitation: Otherwise, the developer faces an *injection attack*, well known to be carried out by SQL query strings [JKK10] or shell commands. If a sanitation is found, the input is killed – hopefully right in time. Most certainly, a tainted input will also touch other variables over its lifetime, passing taintedness over and thus generating new facts that must not reach any sensitive sink point as well.

In the context of MPI RMA, modeling requires a multi-stage fact population: An MPI window always depends on an MPI communicator, whether an explicit one or a global one (`MPI_COMM_WORLD`). Explicit ones need to be made alive first, and since there may be arbitrarily many communicators, any window must be assigned to the correct communicator as otherwise any communicator-related analysis remains impossible.

Fact: `Null`

As introduced by subsection 3.6.1, this type implements $\bot$ and thus acts as the omnipresent singleton fact. It mainly serves these purposes:

1. Generating facts of type `Comm`, `IntConst` and any `Win` that is subject to `MPI_COMM_WORLD`.
2. Tracing values that represent rank values of calls to `MPI_Comm_rank` over `MPI_COMM_WORLD`.
3. Collecting global information that would be stored redundantly if it was collected by each fact individually. For this purpose, a reference to this object is kept by every applicable fact.

Fact: `Comm`

This type represents the lifetime of an explicit MPI communicator. It has three missions:

1. Tracing its own initial `MPI_Comm` handle.
2. Tracing values that represent rank values of calls to `MPI_Comm_rank` over any of its known handles.
3. Generating facts of type `Win` that make use of any of its known handles.

The fact is generated when `Null` hits an MPI method call that creates a new communicator, such as `MPI_Comm_split` or `MPI_Comm_spawn`. It is killed on reaching a destructive method call, including `MPI_Comm_free` or `MPI_Comm_disconnect`.

`Win` facts keep a reference to its generating, explicit communicator, and a `Comm` fact is otherwise free from maintaining relations to other facts but `Null`.

Fact: `Win`

This type represents the lifetime of an MPI window. Considering the topic, it is reasonably rich of tasks to fulfill:

1. Tracing its own `MPI_Win` handle.
2. Maintaining a history of calls and actions that can be linked to this window or any of its related handles. This includes remote actions, local actions, buffer actions and synchronization.
3. Tracing base addresses of locally mapped shared memory, buffer memory and window memory.
4. Managing forks of the history when hitting a conditional branch by generating a child of type `Win` for every branch. The history of the parent is available to a child.
5. Keeping the history locked when traversing edges that have been previously seen by some ancestor `Win` fact as it is the case for loops.
6. Generating facts of type `WinGhost` when being killed.
7. Performing an on-the-fly validation over a set of encountered actions or synchronizations, dying early in case of a violation.

This fact is generated when a `Comm` or `Null` fact meets a call to any of the four window creation methods (cf. section 2.3), or as a child of a preceding `Win` fact on entering a conditional branch or on the exit of loop. It is killed by a call to `MPI_Win_free`.

Fact: `WinGhost`

This is a non-forking, light-weight remainder of a `Win` fact. Being the *ghost* of a window that has been killed earlier, it only keeps track of any subsequent window creation function it encounters, and writes a log entry to the `Null` fact. This information is needed in the second analysis pass in order to correctly group windows of the same communicator globally over the CFG. Thus, it keeps references to both its deceased `Win` fact and the `Null` fact. For reasons of reducing computational complexity, there is an on-the-fly attempt to kill a ghost on hitting `MPI_Comm_free` (or calls alike) since it is not required beyond the lifetime of its communicator. As there must not be any assumptions over the progress of the IFDS problem solution of other facts, the killing is just considered an attempt (cf. subsection 3.6.1) to reduce the overall number of facts subject to the reachability analysis. Otherwise, it peacefully continues to live until the end of the CFG.

Fact: `IntConst`

With more details given in subsection 4.1.7, this fact type represents an integer variable that has been assigned a constant value by an IR instruction such as `store i32 42, i32* %intVariable`,

whether initially, eventually or repeatedly. It serves a limited set of purposes:

1. Tracing its own integer handle.
2. Turning into being *lost* if it observes a write to its memory location without being able to resolve the value to an integer constant. Later on, it may *recover* on succeeding to do so.
3. Similar to `Win`, it also forks as it is required to maintain two different states from that point on, but without writing any history.
4. On the occurrence of a designated set of instructions (comparisons and calls to declared functions such as for MPI), writing a log entry to the referenced `Null` fact about its current value and status for post-IFDS problem analysis steps.

The fact is generated by a `store` instruction writing a constant integer, or as the result of a fork by its parent; it is never killed actively.

A visual overview of the available facts, their relation of generating each other and their requirement of keeping a reference to some other fact during the IFDS problem solution or a at later stage is given by figure 4.2.

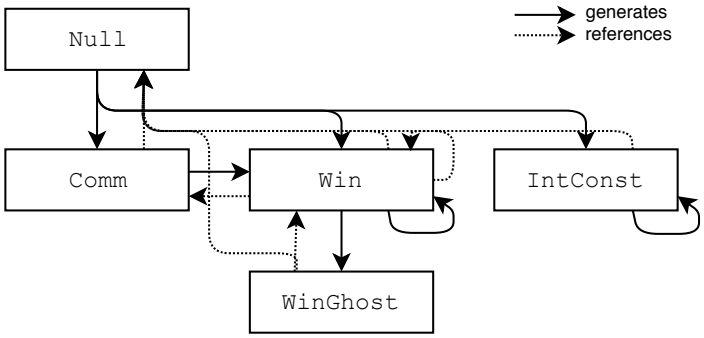


Figure 4.2.: Fact generation and reference diagram.

Tracing – frequently used in the previous fact type descriptions – is by no means a trivial task as it has been implemented to work for the most common cases of writing software:

- *Functions.* A traceable symbol may enter a function and thus requires a mapping into the local body. A traceable symbol may even be generated inside a called function and may return as a value, or written to an address provided by a pointer argument.
- *Global values.* Not only local symbols may be assigned fact handles but also global ones.
- *Structs.* A certain type of fact handles (windows, communicators, integers) may be linked to a precise offset on some given base address.
- *Struct pointers.* For a struct or an object, the base address may just be a pointer to the actual struct that contains the handle. Listing 4.4 demonstrates the C++ equivalent of this problem.
- *Array.* For any kind of memory-based tracing, and in the context of the currently implemented tests presented in section 4.2, contiguous ranges of memory are only traced by uses of their start address.

4.1.4. Path sensitivity

When writing software using MPI, the implementation provides a runner, commonly named `mpirun` or `mpiexec`, that is parameterized by MPI-specific options such as the number of processes

```
1  struct Context { uint32_t i; MPI_Win w };
2  Context c, *p = &c;
3  MPI_Win win;
4
5  MPI_Win_create(..., &win);       // simple
6  MPI_Win_create(..., &(c.w));     // base of c + fixed offset to w
7  MPI_Win_create(..., &(p->w));    // assigned base address + fixed offset to w
```

Listing 4.4: Use of offset-based handles.

to spawn, the path to the executable and its arguments. A common way to make use of the MPI API to orchestrate the work of individual processes is demonstrated by listing 4.5.

```
1  int main (int argc, char **argv) {
2    int rank = -1;
3    MPI_Init(&argc, &argv);
4
5    MPI_Comm_rank(..., &rank);
6
7    if (0 == rank) {
8      MPI_Recv(..., MPI_ANY_SOURCE, ...);
9    } else {
10     MPI_Send(..., 0 /* target rank */, ...);
11   }
12   // ...
13 }
```

Listing 4.5: Obtaining the own MPI rank and heading for the respective tasks.

A call to `MPI_Comm_rank` identifies the process rank under some given communicator, and the acquired, non-negative value may be assigned a role by the developer, commonly using 0 for indicating a *primary*, *master* or *root* process. For the static analysis of MPI RMA, given some `Win` fact, a path-sensitive analysis enables two features:

1. Maintaining a history of notable events, two new histories can be derived and written independently from each other after entering either successor. At the point of killing the fact, it then contains its precise journey through the CFG, allowing a deeper inspection in the post-IFDS problem analysis.
2. Keeping track of the according branch conditions and the actually entered branches (true-case, false-case), the history can be assigned an abstract or precise MPI rank. This enables reasoning on the infeasibility of a path over contradicting rank assumptions.

Implementing a fork of a fact is straight-forward and requires a flow function that kills the incoming fact in either case, generating a descendant for any branch. This behavior applies to facts of `Win` and `IntConst` whereas facts of the other types are mapped identically.

Figure 4.3 visualizes this scenario: Some fact denoted w reaches the node representing a conditional branch. On the edge to its true-successor node, the fact is replaced by w_c, and by $w_{!c}$ in the other case. On reaching the next merge node, both facts happen to coexist in the exploded super-graph.

For the analysis of MPI RMA, a *history* (or *log*) refers to a list of relevant *events* (or *items*; including calls to respective MPI functions, local actions, buffer actions) along one, possibly

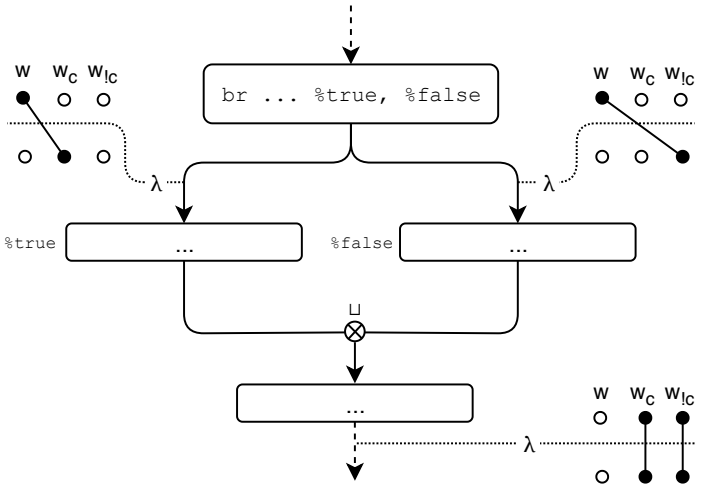


Figure 4.3.: The implementation of path-sensitivity.

infeasible execution *path* through the CFG. For each path, there is one path-sensitive fact for every entering fact of type `Win` or `IntConst`.

4.1.5. Breaking loops

Applying the implementation of path sensitivity to an IFDS problem solution in subsection 4.1.4, without taking any further precautions, this implementation easily violates the requirement of an IFDS problem to expect a finite set of facts in a fairly common scenario: loops.

Doing one step back, loops inside of a CFG require a more general assessment first with respect to a static analysis. Gathering the history of events on the occurrence of a loop, the following semantics is defined, calling it the *exactly-once semantics*:

1. No item is ever recorded more than once per history.
2. Loops with head evaluation (`while` loops) are treated as conditional branches: One fork enters the loop, the other one does not.
3. Loops with foot evaluation (`do while` loops) are treated unconditionally as one iteration is always guaranteed.

In a naive implementation of path sensitivity that fulfills property 1 by scanning its history before appending an event to its log, a fundamental issue remains uncovered: With the design of a flow function that generates a new fact on or after entering the loop body, the subgraph over reachable nodes for such a fact actually includes the node, and hence its outgoing edge that was previously held responsible for generating that fact by hitting the respective flow function, producing a new fact. Thus, a loop will lead to the generation of an infinite number of facts, violating the *finite subset* property of an IFDS problem.

Figure 4.4 demonstrates the problem: For some fact w, the flow function λ_l of the loop-entering edge of the conditional branch will create new fact w_c. For w_c, the set of reachable nodes include `%loop, %head, %exit` and the conditional branch. If λ_l remains agnostic towards the fact w_c of type `Win` or `IntConst`, it generates a fact $w_{c^2}, w_{c^3}, \ldots$, never exiting the cycle that

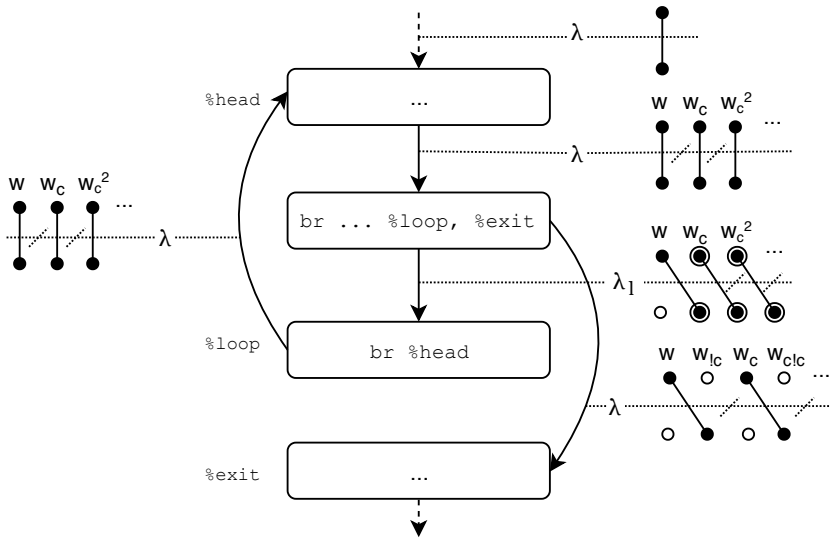


Figure 4.4.: A naive implementation of path-sensitivity for an IFDS problem on loops.

makes the set of facts explode.

This problem is solved by the following mechanism, taking place over the CFG before solving the IFDS problem: If a path-sensitive fact hits the flow function of some back edge, it enters a *fork-locked* state until its reachability analysis hits the edge that finally leaves the associated loop, generating an unlocked successor fact. When in the locked state, no history is written and no flow function will trigger the forking of that fact but apply an identical mapping instead. Any suitable pair of a lock-edge and a respective unlock-edge is precalculated.

Unlock-Map algorithm

Considering an arbitrary depth of nested loops, the *Unlock-Map* algorithm discovers back edges in any CFG and assigns the closest edge to finally leave the loop. This map is then publicly accessible to any path-sensitive fact during the IFDS problem solution to trigger its fork-locked state where necessary.

Briefly, for each function of an LLVM module, the first step consists of a *depth-first search* (DFS) that explores the BBs of a function, creating a preliminary graph structure and discovering its back edges. A *back edge* is an edge that returns to a node for which the search has not yet completed, indicating a cycle, a loop. In the next step, for each edge that has been identified as a back edge, a subgraph is extracted for which the target node of the back edge has no other incoming back edge but the respective edge. Starting from this target node, Tarjan's algorithm for finding *strongly connected component*s (SCCs) [Tar71] is used: An exit of the loop is then defined as an edge that connects the first SCC – the one defined by the cut-off loop subgraph – to another one, the *bridge*.

More formally, the algorithm is described by listing 4.6: Explore-Function-DFS returns a CFG of some given LLVM function by a DFS, Tarjan-SCC performs the original algorithm of Tarjan and assigns every node to a zero-based index of its associated SCC.

```
1  TARJAN-SCC-BRIDGE (G = (V, E, s)):
2    (V_SCC, E_SCC, s) := TARJAN-SCC(G)
3    e := first element of { (n1, n2) ∈ E_SCC | n1.scc = 0 ∧ n2.scc ≠ 0}
4
5    return e
6
7  UNLOCK-MAP (module):
8    map := {}
9
10   for function in module:
11     (V, E, s) := EXPLORE-FUNCTION-DFS(function)
12     E_b := { e ∈ E | e is a back edge }
13
14     for e = (n1, n2) in E_b:
15       G_s := (V, E\(∗, n2) ∪ (n1, n2), n2)
16       e_u := TARJAN-SCC-BRIDGE(G_s)
17       map[e] := e_u
18
19   return map
```

Listing 4.6: The Unlock-Map algorithm.

In the actual implementation of *Unlock-Map*, during the execution of Tarjan's discovery algorithm, the implementation quits early on the discovery of the first suitable bridge. The cost of a single run of Tarjan's algorithm on a graph $G = (V, E)$ is subject to $O(|V| + |E|)$. It is performed for each back edge of the super-set graph of all CFGs, and the set of back edges is a subset of all edges. In total, the cost of this implementation grows no worse than quadratically to the number of edges, i. e. in $O(|E|^2)$.

4.1.6. Window ordering

By the termination of the IFDS problem solver, there exists a set of facts, each one holding a history of events of one path through the code. In order to perform any inter-path or inter-rank analysis, no infeasible pair of `Win` facts must be compared. Any window is said to *coexist* with another window if their lifetimes cannot be ordered: Lifetime ordering is given if on one path, an MPI window a is freed before some window b is created for the same communicator. Two such facts are infeasible to compare

- if the windows they represent do not belong to the same communicator,
- if in their coexistence, the order in which they have been created does not match,
- if they had no coexistence, as the one was created after the other one was freed or
- for the inter-rank analysis, if two facts can be identified as belonging to the same rank.

Assuming the same communicator, the ordering of windows determines the corresponding window of another process. Considering an example code (listing 4.7) for a process that represents a zero-rank, and code that represents any other rank, for rank 0, there are two windows `winA1, winA2`, and `winB1, winB2` for non-zero ranks.

Moreover, in that example, for either branch, two windows are requested. By their order of setup, the counterpart of `winA1` for any non-zero rank is `winB1`. Thus, two windows with a mismatching *setup rank* may ask for communicator-wide considerations over their coexistence but must not be subject to an inter-path analysis.

```
1  if (0 == rank) {
2    MPI_Win winA1, winA2;
3
4    MPI_Win_create(..., &winA1);
5    MPI_Win_create(..., &winA2);
6
7    MPI_Win_fence(0, winA1);
8    MPI_Win_fence(0, winA2);
9    // ...
10 } else {
11   // ... (see right)
12 }
```

Code on rank `0`.

```
1  if (0 == rank) {
2    // ... (see left)
3  } else {
4    MPI_Win winB1, winB2;
5
6    MPI_Win_create(..., &winB1);
7    MPI_Win_create(..., &winB2);
8
9    MPI_Win_fence(0, winB2); // Deadlock
10   MPI_Win_fence(0, winB1);
11   // ...
12 }
```

Code on rank `!= 0`.

Listing 4.7

The `WinGhost` fact has been designed to address the problem of ordering windows in a post-IFDS problem solution step. For being generated immediately after a `Win` fact is killed, for any occurrence of a window-generating MPI call on its reachable nodes, the ghost's passage over this node is registered in the `Null` fact. As a ghost knows its fallen `Win` ancestor, and any `Win` fact knows its generating `Comm` fact, as well as the instruction where its generation occurred, after termination of the IFDS problem solution, the window order can be recovered if at the place of the generation of a `Win` fact the passage of the `WinGhost` fact of a previous window was observed.

To acquire the setup rank of coexisting windows, a similar procedure is used. Since there is a `Win` fact alive at the instruction that generates a subsequent one, no additional fact type is required to cover this situation.

4.1.7. Rank attribution

When performing an analysis over MPI properties, the rank-wise context of a path may be derived from inspecting branch conditions. Whenever a comparison of a value traced from a call to `MPI_Comm_rank` against a constant integer can be identified, valuable information on the associated MPI rank can be elicited. In the step of *rank attribution*, every path is tested for feasibility over non-conflicting rank predicates, as well as the most precise assumption over its associated rank is made.

Considering the practical example of listing 4.8, the following observations are valid:

- A code that executes block `A` has a rank greater than zero.
- Conversely, a path over `B` indicates a rank less or equal to zero. Despite the use of a signed integer, MPI exclusively uses non-negative rank values such that a rank hitting `B` is actually equal to zero.
- Analogously, `C` implies a rank equal to 1, and `D` anything not equal to one.
- Conclusively, a path branching into `B` is not compatible to going through `C` as well.

The most precise rank assumptions are derived from paths crossing `B` (equals 0) or `C` (equals 1); a contradiction over the infeasibility of a path through `B` or `C` allows the elimination of that

```
1   int rank = -1, root = 0;
2   MPI_Comm_rank(..., &rank);
3
4   if (rank > root) {
5     // A
6   } else {
7     // B
8   }
9
10  if (1 == rank) {
11    // C
12  } else {
13    // D
14  }
```

Listing 4.8: Constraints on the code path over the MPI rank.

path. There is less precision over a path through `A` (greater than 0) and `C` (not equal to 1): This path has a rank greater or equal to 2.

A *rank predicate* is defined over a tuple relation $P = \mathbb{N}_0 \times Op$ of the non-negative MPI rank and $Op = \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \bot\}$ being a set of operators meaning same as their C/C++ equivalents whereas $\bot$ denotes *not resolvable*. By exploiting the non-negativity of MPI ranks, for some rank predicate, the following equivalent simplifications are made to the predicate:

$$
\begin{aligned}
(0, \texttt{<=}) &\mapsto (0, \texttt{==}) \\
(0, \texttt{!=}) &\mapsto (0, \texttt{>}) \\
(1, \texttt{<}) &\mapsto (0, \texttt{==})
\end{aligned}
$$

A mapping $P \times P \to \{\bot, \not\bot\}$ helps identifying a *contradiction* of two given rank predicates:

$$
\begin{aligned}
&\text{with } a \neq b\text{:} \\
&\qquad (a, \texttt{==}), (b, \texttt{==}) \mapsto \bot \\
&\text{with } a < b\text{:} \\
&\qquad (a, \texttt{<}), (b, \texttt{>}) \mapsto \bot \\
&\qquad (a, \texttt{<=}), (b, \texttt{>}) \mapsto \bot \\
&\qquad (a, \texttt{<}), (b, \texttt{>=}) \mapsto \bot \\
&\qquad (a, \texttt{<=}), (b, \texttt{>=}) \mapsto \bot \\
&\qquad (a, \texttt{<}), (b, \texttt{==}) \mapsto \bot \\
&\qquad (a, \texttt{<=}), (b, \texttt{==}) \mapsto \bot \\
&\qquad (a, \texttt{==}), (b, \texttt{>}) \mapsto \bot \\
&\qquad (a, \texttt{==}), (b, \texttt{>=}) \mapsto \bot
\end{aligned}
\qquad
\begin{aligned}
&\text{with } a \leqslant b\text{:} \\
&\qquad (a, \texttt{==}), (b, \texttt{>}) \mapsto \bot \\
&\qquad (a, \texttt{<=}), (b, \texttt{>}) \mapsto \bot \\
&\qquad (a, \texttt{<}), (b, \texttt{>}) \mapsto \bot
\end{aligned}
$$

Analogously for $a > b$ and $a \geqslant b$.

with $a = b$:

$$
\begin{array}{rlllll}
(a, \texttt{>}), (b, \texttt{<=}) & \mapsto \bot & (a, \texttt{==}), & (b, \texttt{>}) & \mapsto \bot \\
(a, \texttt{>}), (b, \texttt{<}) & \mapsto \bot & (a, \texttt{==}), & (b, \texttt{<}) & \mapsto \bot \\
(a, \texttt{>}), (b, \texttt{==}) & \mapsto \bot & (a, \texttt{!=}), & (b, \texttt{==}) & \mapsto \bot \\
(a, \texttt{>=}), (b, \texttt{<}) & \mapsto \bot & & & \\
(a, \texttt{<}), (b, \texttt{>=}) & \mapsto \bot & & & \\
(a, \texttt{<}), (b, \texttt{>}) & \mapsto \bot & & & \\
(a, \texttt{<}), (b, \texttt{==}) & \mapsto \bot & & & \\
(a, \texttt{<=}), (b, \texttt{>}) & \mapsto \bot & & & \\
(a, \texttt{==}), (b, \texttt{!=}) & \mapsto \bot & & &
\end{array}
$$

everything else $\mapsto \not\perp$

For two rank predicates that are not in a contradiction, there is a mapping $P \times P \to P$ to extract the one of the higher *precision*, i. e. the one representing a smaller set of ranks, or to merge them into a more precise predicate:

with $op_a \neq$ `==`:

$$(a, op_a), (b, \texttt{==}) \mapsto (b, \texttt{==})$$

with $op_a = \bot, op_b \neq \bot$:

$$(a, op_a), (b, op_b) \mapsto (b, op_b)$$

with $a = b$:

$$
\begin{array}{ll}
(a, \texttt{<=}), (b, \texttt{<}) & \mapsto (b, \texttt{<}) \\
(a, \texttt{>=}), (b, \texttt{>}) & \mapsto (b, \texttt{>}) \\
(a, \texttt{<=}), (b, \texttt{>=}) & \mapsto (a, \texttt{==}) \\
(a, \texttt{>=}), (b, \texttt{<=}) & \mapsto (a, \texttt{==})
\end{array}
$$

otherwise:

$$p_a, p_b \mapsto p_a$$

Technically, these mappings are applied to the LLVM IR. With C code shown by listing 4.8, the corresponding IR is partially shown by listing 4.9: There is one symbol representing the test value of root rank zero (`%7`) and another one that will be assigned the communicator's rank (`%6`). Later, they are being loaded into `%11, %12` for testing their greater-than relation (`rank > root`) by `icmp sgt i32 %11, %12`. For an execution path entering label `14`, the rank predicate is $(0, \texttt{>})$, contrary to $(0, \texttt{<=})$ for the other path.

For the given IR, the work is distributed as following across different fact types:

- `Null`. For calling `MPI_Comm_rank` on `MPI_COMM_WORLD`, the `Null` fact is obligated to keep track of `%6`. For other communicators, the corresponding `Comm` fact is in charge.
- `IntConst`. As there is a `store` instruction assigning a constant (`0`) to a symbol (`%7`), an `IntConst` fact is generated that will log its value at any significant point, such as the mentioned `icmp` instruction.

```
; ...
%6 = alloca i32, align 4
%7 = alloca i32, align 4
store i32 0, i32* %7, align 4
; ...
%10 = call i32 @MPI_Comm_rank(i32 1140850688, i32* %6)
%11 = load i32, i32* %6, align 4
%12 = load i32, i32* %7, align 4
%13 = icmp sgt i32 %11, %12
br i1 %13, label %14, label %17
```

Listing 4.9: Testing for MPI ranks in LLVM IR.

- `Win`. A `Win` fact remembers all passed conditional branches and the successor node that has been entered. From the conditional branch instruction, the comparison being used can be identified for trivial cases.

The procedure is given in more detail by listing 4.10: Simplify-Predicate, Contradicting-Predicate and More-Precise-Predicate describe the mappings mentioned earlier in this subsection. Get-Positive-Predicate extracts the positive predicate with respect to the branch that has been entered by *item*. LLVM assists in that step by providing `llvm::CmpInst::getInversePredicate` for properly inverting the predicate that appears in the IR.

```
Rank-Attribute (path):
  predicate := (0, ⊥)

  for item in path.items:
    if item is a conditional llvm::BranchInst:
      candidate_predicate := Get-Positive-Predicate(item)
      candidate_predicate := Simplify-Predicate(candidate_predicate)

      unless Contradicting-Predicates(predicate, candidate_predicate):
        predicate := More-Precise-Predicate(predicate, candidate_predicate)

  return predicate
```

Listing 4.10: Rank attribution algorithm

Simplify-Predicate, Contradicting-Predicate and More-Precise-Predicate are bound by a constant number of steps whereas Get-Positive-Predicate requires a constant number of symbol searches concerning the involved facts, the most expensive one being the test for the ownership of the symbol for each fact of type `IntConst` that is alive at the point of comparison. A rank predicate is only accepted if the associated integer is met over all paths.

Let I denote the set of all `IntConst` facts. Unless path-sensitivity for `IntConst` has been disabled by the user, $|I| \in O(2^{|V|})$; $|I| \in O(|V|)$ otherwise. Let L_p denote the list of events of some path p, then the overall costs to determine the rank predicate for that path p is subject to $O(|L_p| \cdot |I| \cdot log(|V|))$ – the test for the ownership of a symbol for one fact is a logarithmic look-up.

4.2. Property tests

For examining properties of MPI RMA for some given IR module, there are mainly two phases that suit different tests. This circumstance is due to the different availability of information but also for better managing computational complexity:

1. *IFDS problem solution tests*: As every `Win` fact collects items for its history, a limited set of tests for violating the use of MPI RMA can be made on the fly, for example using `MPI_Put` without any prior call to enter the corresponding epoch.
2. *Post-IFDS problem solution tests*: Some tests cannot be performed until the IFDS problem solver has terminated completely. Tests of this phase include:
   a) *Intra-path tests*: This class of tests does not require correlating a path to any other path. It mainly performs tests that suffice a linear scan over the history.
   b) *Inter-path tests*: In this class, as a first step, window paths must be grouped by their lifetime epochs, communicator and setup rank. Theses tests do not require any knowledge of ranks but audit communicator- or window-wide aspects, such as a proper use of barriers, fences and the concordance of flags.
   c) *Inter-rank tests*: To perform this class of tests, each path must undergo the rank attribution procedure first. Then, with the help of `IntConst` for collecting possible values of target ranks in the communication calls, the interaction between two or more paths can be examined.

Currently, identifying a violation in a path will exclude it from any following analysis. For this reason, the order of post-IFDS problem solution test classes has been arranged by their increasing computational complexity.

For this section and the subsequent ones, the following classes are defined for grammar listings and PCRE-style[1] regular expressions:

$c_{\mathrm{F,G,P}}$: legal comm. calls for fencing (F), GATS (G) and PTS (P)
f : `MPI_Win_fence` without assertion flags
$f_{\mathrm{NP,NS}}$: `MPI_Win_fence` with assertion flags (cf. figure 2.3a, p. 13)
$g_{\mathrm{C,W}}$: `MPI_Win_complete` (C) or `MPI_Win_wait` (W)
$g_{\mathrm{P,S}}$: `MPI_Win_post` (P) or `MPI_Win_start` (S)
p_{FS} : `MPI_Win_flush`
$p_{\mathrm{L,LA}}$: `MPI_Win_lock` (L) or `MPI_Win_lock_all` (LA)
$p_{\mathrm{UL,ULA}}$: `MPI_Win_unlock` (UL) or `MPI_Win_unlock_all` (ULA)
x : `MPI_Win_free`

4.2.1. IFDS problem tests

The different MPI RMA synchronization modes, approximately shown by figure 2.3a, figure 2.3b (p. 13) and figure 2.4 (p. 14), can be tested for violations at the execution time of the

[1] PCRE(3) man page – `https://www.pcre.org/pcre.txt`, last seen 2018-12-16.

IFDS problem solver. The problem of determining the set of allowed operations on an MPI window in a given context can be considered an instance of a *typestate analysis* [Str83].

By carefully merging the FSMs into one super-FSM that allows a window fact to enter all legal states over its lifetime, for a subset of calls to relevant MPI methods, violations can be detected easily. The requirement to count a number of locks and to await the correct number of unlocks requires the super-FSM to turn into a more powerful PDA. A valid history is an accepted word of the language defined by a context-free grammar. The high-level components of this grammar are described by the entry rule, the different window modes and the terminator symbol:

$$\begin{aligned} S &\to O \\ O &\to F^S | G_P | G_S | L | L_A | X \\ X &\to x \end{aligned}$$

For the fence mode, these production rules are given:

$$\begin{aligned} F^S &\to f F^\# | f_{\mathrm{NP}} F^\# | f_{\mathrm{NS}} F^\# | f_{\mathrm{NP|NS}} F^\# & F_{\mathrm{NP}} &\to f_{\mathrm{NP}} C_{\mathrm{F}} | f_{\mathrm{NP}} F^* \\ F^* &\to F^\# | O & F_{\mathrm{NS}} &\to f_{\mathrm{NS}} F^* \\ F^\# &\to F | F_{\mathrm{NP}} | F_{\mathrm{NS}} | F_{\mathrm{NP|NS}} & F_{\mathrm{NP|NS}} &\to f_{\mathrm{NP|NS}} F^* \\ F &\to f C_{\mathrm{F}} | f F^* & C_{\mathrm{F}} &\to c_{\mathrm{F}} C_{\mathrm{F}} | F | F_{\mathrm{NS}} \end{aligned}$$

The history for GATS is subject to these rules:

$$\begin{aligned} G_P &\to g_{\mathrm{P}} G_{P*} & G_{S*} &\to C_{\mathrm{G}} g_{\mathrm{C}} O | g_{\mathrm{P}} G_{PS} \\ G_S &\to g_{\mathrm{S}} G_{S*} & G_{P*} &\to g_{\mathrm{W}} O | g_{\mathrm{S}} G_{PS} \\ G_{PS} &\to g_{\mathrm{W}} G_{S*} | C_G g_{\mathrm{C}} G_{P*} & C_G &\to c_{\mathrm{G}} C_G | \epsilon \end{aligned}$$

Lastly, the production rules for PTS:

$$\begin{aligned} L &\to p_{\mathrm{L}} L^* p_{\mathrm{UL}} O & C_P &\to c_{\mathrm{P}} C_P | M_{\mathrm{FS}} | \epsilon \\ L^* &\to C_P L^* | p_{\mathrm{L}} L^* p_{\mathrm{UL}} & M_{\mathrm{FS}} &\to p_{\mathrm{FS}} C_P \\ L_A &\to p_{\mathrm{LA}} C_P p_{\mathrm{ULA}} O \end{aligned}$$

The invalidity of the history of one code path is determined by two cases: by failing to submit to the accepted language, or by not reaching the termination symbol. In the grammar, a subset of the input alphabet is ignored – or *always accepted* – for its postponed relevance. This set includes buffer actions, local actions, a subset of MPI methods (e. g. `MPI_Barrier` or `MPI_Comm_free`) and branch instructions.

Despite formally using a grammar, the implementation is written over a hard-coded super-FSM with simple algorithmic extensions, sacrificing any kind of genericness that otherwise requires tools such as the *CYK algorithm* [You66]. This benefits the ability to integrate the test online at the IFDS problem solution time.

Unless the tool has been built against the correct MPI implementation, the test for the semantics imposed by the flags of `MPI_Win_fence` is discarded and the grammar shrinks respectively.

4.2.2. Intra-path tests

For the set of intra-path tests, it is sufficient to test the history of a single path for properties. It is not required to consider information about its setup rank or order alongside other paths, to take communicator-wide aspects into account or to have a resolved rank predicate.

Illegal memory requests

This test scans for the illegal use of queries for shared memory on windows that are not of the shared flavor, as well as for attempts to dynamically modify memory of windows that have not been initialized for the dynamic flavor. Three new alphabet classes are introduced:

$$\begin{aligned} m &: \texttt{MPI\_Win\_attach} \text{ or } \texttt{MPI\_Win\_detach} \\ s &: \texttt{MPI\_Win\_shared\_query} \\ w_{\mathrm{A,C,D,S}} &: \texttt{MPI\_Win\_allocate } (\mathrm{A}),\ \texttt{MPI\_Win\_create } (\mathrm{C}), \\ &\quad \texttt{MPI\_Win\_create\_dynamic } (\mathrm{D}) \text{ or } \texttt{MPI\_Win\_allocate\_shared } (\mathrm{S}) \end{aligned}$$

Given two regular expressions:

$$\hat{}[w_{\mathrm{A}}|w_{\mathrm{C}}|w_{\mathrm{D}}].^*s \qquad \hat{}[w_{\mathrm{A}}|w_{\mathrm{C}}|w_{\mathrm{S}}].^*m$$

If the prefix of a history is matched by any of the given regular expressions, a violation has been identified. In the example of listing 4.11, for three windows of different flavors, there are two violations that can be detected: The illegal use of `MPI_Win_attach` by `winA`, as well as `MPI_Win_shared_query` by `winB`.

```
1  MPI_Win winA, winB, winC;
2
3  MPI_Win_create(..., &winA);
4  MPI_Win_create_dynamic(..., &winB);
5  MPI_Win_allocate_shared(..., &winC);
6
7  MPI_Win_attach(winA, ...);
8  MPI_Win_shared_query(winB, ...);
```

Listing 4.11: Illegal memory management (`winA`), shared memory query (`winB`).

This test is implemented by a fairly small FSM.

Unsafe buffer actions

A mistake that is easily made when writing MPI RMA code is the non-synchronized use of buffer actions. Considering the code of listing 4.12, there are two mistakes that lead to a conflict in both cases: An attempt to read from or to write to `buffer` before the synchronization by `MPI_Win_fence` will lead to a data race.

```
1  // ...
2  MPI_Put(buffer, ..., winA);
3  buffer[0] = 42;
4  MPI_fence(..., winA);
5
6  MPI_Get(buffer, ..., winB);
7  uint32_t value = buffer[0];
8  MPI_fence(..., winB);
```

Listing 4.12: Non-synchronized buffer actions.

Although terminating the communication epoch by `MPI_Win_fence` correctly twice, the write to `buffer` is not consistent to `MPI_Put` as well as the read from from `buffer` right after `MPI_Get`. Another set of classes will be referenced:

$$b : \text{buffer actions}$$
$$f_* : [f|f_{\mathrm{NP}}|f_{\mathrm{NS}}|f_{\mathrm{NP|NS}}]$$

Given the class b, with (α) binding the previous input to a memory buffer by a symbol, regular expressions can be created to test for unsafe buffer actions:

$$c_{\mathrm{F}}(\alpha)[\hat{}f_*]^*b(\alpha) \quad c_{\mathrm{G}}(\alpha)[\hat{}g_{\mathrm{C}}]^*b(\alpha) \quad c_{\mathrm{P}}(\alpha)[\hat{}p_{\mathrm{FS}}|p_{\mathrm{UL}}|p_{\mathrm{ULA}}]^*b(\alpha)$$

Rephrasing the expressions, they test whether there is a remote action on some memory buffer α not followed by the appropriate synchronization before a buffer action on α.

The state of the implementation, an FSM with additional effort to scan for matching memory symbols, turns this test into a falsification: Classes p_{UL} and p_{FS} include MPI methods that synchronize a particular rank for a preceding remote action to that rank, for example `MPI_Win_unlock` or `MPI_Win_flush`. As the test lacks the ability to correlate communication and synchronization ranks at this stage, the absence of synchronization of class p_{UL} and p_{FS} indicates the violation of the consistency property. Conversely, its presence must not tempt to assume correctness.

4.2.3. Inter-path tests

For this class of tests, paths need be grouped in advance to only test feasible pairs of paths against each other (cf. subsection 4.1.6).

Window flavor compatibility

When using windows by different flavors, if one window is initialized for the shared flavor, all matching windows must be subject to the shared flavor, too [For15, section 11.2.3]. The code of listing 4.13 demonstrates such a violation as the window of rank `0` is of the shared flavor while any other window is a statically-flavored one, assuming the same underlying communicator.

Let W be the set of properly grouped `Win` facts, with each fact knowing its window flavor, then the test can be described formally by first-order logic:

```
1   MPI_Win win;
2   int rank = -1;
3
4   MPI_Init(...);
5   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
6
7   if (0 == rank) {
8     MPI_Win_allocate_shared(..., &win);
9   } else {
10    MPI_Win_create(..., &win);
11  }
```

Listing 4.13: Illegal setup of MPI windows.

$$\exists w \in W.w \text{ uses shared flavor} \Rightarrow \forall w \in W.w \text{ uses shared flavor}$$

Deadlocks

In the context of MPI, and MPI RMA in particular, incautiously using `MPI_Barrier` and `MPI_Win_fence` summons deadlocks that may move a whole communicator to an unfortunate end. Deadlocks over these synchronization calls occur unless the sequence of these calls is the same for all windows of the same setup rank. Although deadlocks may as well occur for other reason that are not related to MPI at all, any sub-communicator deadlock bubbles towards the communicator scope if this prevents all processes to eventually reach a call to `MPI_Barrier`.

This test focuses on the use of `MPI_Barrier` and `MPI_Win_fence`, necessarily on a communicator-wide level if, and only if, a call to `MPI_Barrier` occurs at the lifetime of a window, as execution paths can be free from MPI windows at all despite making use of `MPI_Barrier` concurrently.

An example for a deadlock over falsely ordering window fences was previously shown by listing 4.7 (p. 43). Analogously, a deadlock is doomed to occur over using barriers as shown by the source code of listing 4.14 as the calls to `MPI_Win_fence` and `MPI_Barrier` are not ordered consistently across the ranks.

```
1   if (0 == rank) {
2     MPI_Win win;
3     MPI_Win_create
4       (..., MPI_COMM_WORLD, &win);
5
6     MPI_Win_fence(0, win);
7     MPI_Barrier(MPI_COMM_WORLD);
8   } else if (1 == rank) {
9     // ... (see right)
10  } else {
11    MPI_Barrier(MPI_COMM_WORLD);
12  }
```

Code on rank `0`.

```
1   if (0 == rank) {
2     // ... (see left)
3   } else if (1 == rank) {
4     MPI_Win win;
5     MPI_Win_create
6       (..., MPI_COMM_WORLD, &win);
7
8     MPI_Barrier(MPI_COMM_WORLD);
9     MPI_Win_fence(0, win);
10  } else {
11    MPI_Barrier(MPI_COMM_WORLD);
12  }
```

Code on rank `1`.

Listing 4.14: A deadlock situation.

Assuming the absence of any preceding windows before splitting the execution path by the MPI rank, the call to `MPI_Barrier` in line 11 will nowhere appear in any history as collecting

calls to `MPI_Barrier` is done by `Win` facts.

By the given design of the IFDS problem fact domain, this test has its natural limitation: By closely inspecting listing 4.7 (p. 43) again, the problem design will generate at least four `Win` facts that can be grouped pairwise: {`winA1, winB1`} and {`winA2, winB2`}. As a `Win` fact is supposed to collect an item for the history only if it belongs to that window without doubt, it steps over calls to MPI methods that it does not accept as the own ones. Thus, either group remains blind for one call to `MPI_Win_fence`.

Nonetheless, there are some more obvious things left to test: Given some two or more paths grouped by the same setup rank, unless all paths have the same number of calls to `MPI_Win_fence`, at least one will get stuck while another one will terminate. The test can be globally extended to calls to `MPI_Barrier(MPI_COMM_WORLD)` with `MPI_COMM_WORLD` provided as a constant parameter. More precisely, this test is able to identify two deadlock criteria:

1. For two or more facts, there is at least one that has a different number of calls to `MPI_Win_fence` and `MPI_Barrier(MPI_COMM_WORLD)` than some other one of the same group.
2. One fact awaits all other facts to arrive at `MPI_Win_fence` whereas some other fact expects them to reach `MPI_Barrier(MPI_COMM_WORLD)`. If `MPI_COMM_WORLD` cannot be proven as the exact parameter everywhere, this step of the test is discarded for undecidability.

The implementation of this test can be modeled best as an ad-hoc *place/transition net* (P/T net). Let a fact represent a token, and let each occurrence of a synchronization be a transition. A transition's preset and postset contain as many disjoint places as there are facts in the group.

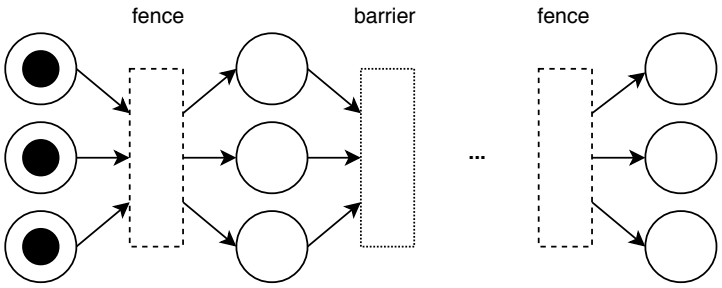


Figure 4.5.: A path group free of a specific class of deadlocks.

For a group of three paths that is free of the kind of deadlocks that this test attempts to identify, an according P/T net is shown by figure 4.5. In contrast, if the path group violates any of the given criteria, its P/T net is partially modeled by *dead* transitions that will never put any fact-token towards the termination place. The two cases for such dead transitions are shown by figure 4.6.

The implementation of the test spots such a deadlock if there are no tokens left to move while not every token has reached its terminating destination. Let W denote the group of `Win` facts for the group under analysis, and let $n = \max\{\ |w.\text{history}|\ |w \in W\}$ denote the longest history provided by the facts. Then the number of steps to determine a deadlock is bound by $O(n \cdot |W|)$.

A source of false positives for this tests lies in the inability to further group paths by mutually exclusive sections, for example by run time options changing the set of `MPI_Win_fence` calls consistently: For every conditional branch, all path histories must exhibit the same number of calls.

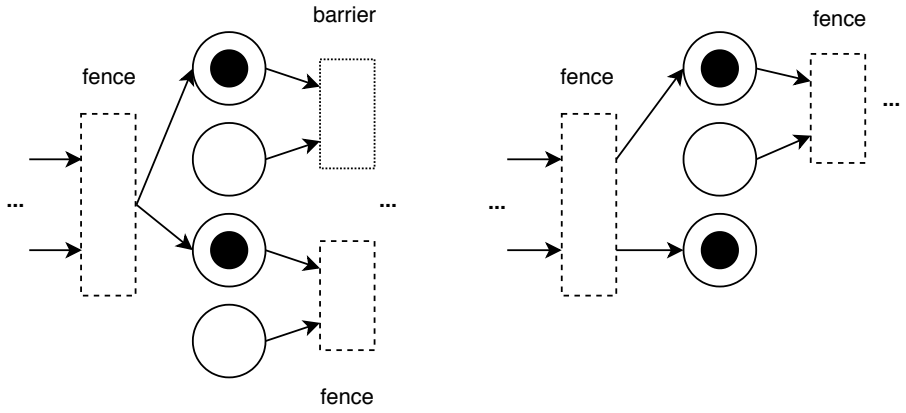


Figure 4.6.: Deadlock by fence/barrier divergence (left), and by the number of transitions (right).

Fence flags

This test expects every fact and its respective group to have undergone the previous test for getting stuck over synchronization calls successfully, as well as having the tool compiled against the respective MPI implementation that has been used for the program of the IR.

Previously described on p. 13, certain assertion flags of `MPI_Win_fence` must be set the same across all paths of a window group that enter that fence. A counter-example is shown by listing 4.15 that shows one call to `MPI_Win_fence` using the `MPI_MODE_NOSUCCEED` while its matching call on the other rank case omits that flag.

```
1   MPI_Win win;
2   int rank = -1;
3
4   MPI_Win_create(..., &win);
5   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
6
7   if (0 == rank) {
8     MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
9   } else {
10    MPI_Win_fence(0, win);
11  }
12  // ...
```

Listing 4.15: Mismatching flags for `MPI_Win_fence`.

Under the assumption of a previously positive test for correctly aligned synchronization calls, let n_{fences} denote the number of calls to `MPI_Win_fence` of all paths of the current group W, with each $w \in W$ having a list named *fenceFlags*, ordered by the occurrence of fence calls of that path, containing the relevant MPI flags, filtered for `MPI_MODE_NOPRECEDE` and `MPI_MODE_NOSUCCEED`. The test then checks for the following property, expressed by first-order logic:

$$\forall i \in \{0..n_{\text{fences}}\}. \underset{w \in W}{=} (w.\text{fenceFlags}[i])$$

The implementation iterates over all paths and compares the union of the mentioned flags for every i-th call to `MPI_Win_fence`, expecting equivalence at each call across all paths. Again, the number of steps to test this property is bound by $O(n \cdot |W|)$.

4.2.4. Inter-rank tests

For the analysis of inter-rank properties, every group must pass the rank attribution procedure in advance, described by subsection 4.1.7. `Win` facts that represent paths that are proven to be infeasible over contradicting tests of the associated MPI rank predicates are filtered here.

Epoch matching

With the ability at the horizon to discover conflicts across paths, under the assumption that every previous test has been passed successfully, a mechanism must be installed to generally ensure the following properties within a group of window paths:

1. Paths of the same rank, i. e. with the same assigned rank attribute $(a, ==)$, must not be compared to each other.
2. For two or more paths, only actions that meet inside the same chronological epoch must be analyzed.

Assuming these properties to hold, the following properties can be derived for testing:

1. If one path has no counterpart epoch found in any other legal path, this may be an error for certain window modes: For a call to `MPI_Win_start`, there must be a corresponding call to `MPI_Win_post`, as well as a call to `MPI_Win_lock` must not stand alone – not assuming any kind of self-locking.
2. If there exist counterparts for one path with respect to an epoch, there must be a compatible epoch among those: A call to `MPI_Win_post` cannot be solely accompanied by calls to `MPI_Win_lock` on all other paths.

An example is given by listing 4.16: Assuming no superior control flow around the given snippets of code, any process of rank predicate $(0, ==)$ will enter a GATS access epoch while any complementary process attempts to lock the root process. The conditional branching over `someCondition` is subordinate to the rank comparison: It is obvious that for a rank predicate of type $(a, ==)$, the two forks `A` and `B` must not be examined against each other since they cannot exist concurrently under an MPI execution.

Let the idea of the algorithm described by listing 4.17 be rephrased: For each path $w_i \in W$, take every other feasible path $w_j \in W$ and assign each item of w_i to any applicable history item of w_j. The assignment is grouped by matching the epochs of the two paths as far as they are compatible to each other.

The method Get-Epoch-Of-Path used in the algorithm of listing 4.17, as well as the *epochs* attribute of some given path w_j have the same underlying mechanism: By taking table 2.1 (p. 17) into account for counting the i-th epoch over the i-th occurrence of an ordering call (except request-induced `MPI_Wait` and its derivatives), the epoch e_i is part of a tuple, accompanied by its symbol or name $name_i$, and the set $I_{j,i}$ of instructions of the history following until

```
1  MPI_Barrier(comm);
2
3  if (0 == rank) {
4    MPI_Win_start(..., win);
5
6    if (someCondition) { // A }
7    else { // B }
8    // ...
9  } else {
10   // ... (see right)
11 }
```

Code on rank `0`.

```
1  MPI_Barrier(comm);
2
3  if (0 == rank) {
4    // ... (see left)
5  } else {
6    MPI_Win_lock(..., 0, 0, win);
7    // ...
8  }
```

Code on rank `!= 0`.

Listing 4.16: Illegal setup of epochs.

the next ordering call. For the same reasons that were previously described in the context of the deadlock prediction (p. 51), only barriers as calls to `MPI_Barrier(MPI_COMM_WORLD)` are considered for ordering actions.

```
1  Epoch-Matching (W):
2    mappings := {}
3
4    for w_i in W:
5      assignments := ∅
6
7      for w_j in W\{w_i}:
8        (a_i, op_i) := w_i.rankPred
9        (a_j, op_j) := w_j.rankPred
10
11       if op_i == and (a_i, op_i) = (a_j, op_j):
12         continue
13
14       for (e_k, name_ik, I_i,k) in w_i.epochs:
15         I_j,k, name_jk := Get-Epoch-Of-Path(e_k, w_j)
16
17         if Epochs-Are-Compatible(name_ik, name_jk):
18           for l_k in I_i,k:
19             for m_k in I_j,k:
20               assignments := assignments ∪ {(l_k, (w_j, m_k))}
21
22     mappings := mappings ∪ {(w_i, assignments)}
23
24   return mappings
```

Listing 4.17: Epoch matching.

The method Epochs-Are-Compatible tests whether two given epochs are mutually exclusive or not.

For every path w_i, the algorithm generates a tuple $(w_i, I_i \times (W \backslash \{w_i\}, I_{\neq i}))$ such that each of its items in I_i is mapped to all applicable items of the other paths. The algorithm is quite costly, considering that $|W| \in O(2^b)$ – b being the number of conditional branch instructions –, with the number of steps to cross-assign all items of all path histories being bound to $O((n \cdot |W|)^2)$.

Once the cross-path assignment of instructions has been completed, all path assignments can be tested for the properties mentioned at the beginning of this subsection, with the number of steps to test any property bound by $O(n \cdot |W|)$ per path.

Concurrent local writes

Even more assumptions can made by making use of `IntConst` facts if their value meets over all paths: The target rank of remote actions, specified by an argument of methods such as `MPI_Get` or `MPI_Win_shared_query` combined with following local actions on shared memory can possibly be acquired by scanning the available `IntConst` facts at the occurrence of such actions. Unless the target rank, expressed as $(a, ==)$, and some other path's rank attribute of this epoch can be proven as a contradiction – in the same way as presented by subsection 4.1.7 – this path can be considered a potential target of that remote action.

```
1  const int rootRank = 0;
2
3  if (rootRank == rank) {
4    uint8_t *memory = ...;
5    MPI_Win_create(memory, ..., &win);
6    MPI_Win_fence(..., win);
7
8    memory[0] = 42;
9
10   MPI_Win_fence(..., win);
11   // ...
12 } else {
13   // ... (see right)
14 }
```

Code on rank `0`.

```
1  const int rootRank = 0;
2
3  if (rootRank == rank) {
4    // ... (see left)
5  } else {
6    uint8_t *buffer = ...;
7    MPI_Win_create(..., &win);
8    MPI_Win_fence(..., win);
9
10   MPI_Put(buffer, ..., rootRank, ..., win);
11
12   MPI_Win_fence(..., win);
13   // ...
14 }
```

Code on rank `!= 0`.

Listing 4.18: Concurrent local write to memory window.

Portable MPI RMA code should assume the presence of a separate window memory model. Under this memory model, there must not exist any local write to the private copy of the window if there is a remote action targeting the public copy of the window in the same epoch, regardless of details to overlapping of the targeted memory regions.

An example of such a situation is given by listing 4.18: Between two fences, the process of rank `0` locally writes a value to `memory`, hence modifying the private window copy. At the same time, any complementary rank performs a remote write action targeting rank `0`. Under the epoch matching algorithm described by the previous subsection, there will be the following assignments to the call of `MPI_Put` from any fact $w_{\neq 0}$, assuming w_0 being the only fact acquiring the rank predicate $(0, ==)$:

$$\begin{aligned} \texttt{MPI\_Put} \rightarrow \{&(w_0, \texttt{MPI\_Win\_fence}),\\ &(w_0, \texttt{memory[0] = 42}),\\ &(w_0, \texttt{MPI\_Win\_fence})\} \end{aligned}$$

From the call of `MPI_Put`, in this case, the target rank can be extracted precisely: $(0, ==)$. Since it is not only contradiction-free to the rank predicate of w_0 but even a perfect match, the assignments of that rank can be safely scanned for problematic actions: `memory[0] = 42`. If there were only synchronization calls or other actions that had no impact on the private copy, there would not be any violation.

To scan all the assignments to a remote action for the inclusion of a local action, the number of assignments to scan is not larger than the number of assignments being made in total by the epoch matching algorithm.

4.3. Summary

In this chapter, multiple property tests for MPI RMA features have been modeled that are carried out during and after the IFDS problem solution. An overview of the steps starting at the IFDS problem solution and later is given by table 4.1 in the order of execution, given the following symbols:

(V, E) : vertices, edges of all CFGs
b : the total number of conditional branch instructions
n : the largest number of history items
w : the total number of paths

Property / Step	**Stage**	**Complexity**
window mode API compliance	IFDS problem	$O(2^{3b})$
window path grouping	-	$O(w \cdot log(w))$
memory queries by mode	intra-path	$O(w \cdot n)$
unsafe buffer use	intra-path	$O(w \cdot n)$
window flavor compatibility	inter-path	$O(w)$
deadlock detection	inter-path	$O(w \cdot n)$
fence flag matching	inter-path	$O(w \cdot n)$
rank attribution	-	$O(2^{\|V\|})$ or $O(\|V\| \cdot log(\|V\|) \cdot n \cdot w)$
epoch matching	inter-rank	$O(2^{2b})$
concurrent local write tests	inter-rank	as *epoch matching*

Table 4.1.: MPI RMA synchronization calls

Reading the MPI specification, there are other things of interest to test in a static analysis: Some properties have a more global meaning, such as being subject to communicator groups and calls to `MPI_Barrier`. Other properties are closely related to MPI RMA and relatively easy to violate unless the required degree of precaution has been applied during development, for example writing concurrently into overlapping memory of a target window.

Nonetheless, tests for these properties are not trivial to model, or computationally cheap to perform, or both, even for a falsification. In section 8.1, a more comprehensive list of current limitations and their justification is given, as well as open topics and possible approaches to address these issues.

The subsequent chapter presents an evaluation of the IFDS problem modeling and the tests on non-trivial source code.

5. Evaluation

After modeling property tests for MPI RMA as an IFDS problem and reusing the results for follow-up tests, this chapter covers the evaluation of the tool on non-trivial code, both for scalability and suitability, and presents approaches to run time optimization.

After designing the algorithms and tests for MPI RMA properties, the next step is to perform experiments and observe the overall performance and quality of the tool. For the IFDS problem solution and some of the post-IFDS problem tests, combined with the overall potential number of `Win` and `IntConst` facts, in theory, the computational cost is quite extraordinary.

In the first part of this chapter, there will be simple types of programs that have been designed to assess the base costs of using Phasar, and to confirm the run time growth of the naive IFDS problem solution over a steady increase of the number of conditional branches. From the perspective of both technical and modeling aspects, optimization techniques and their implementation will be discussed. In the second part, the tool will be used in an experiment on two non-trivial MPI RMA code bases that will lead to decisions covered by the next chapter.

5.1. Scalability

For estimating the computational complexity of the IFDS problem model, let the following variables be given:

b : the number of conditional branch instructions
m : the number of calls to methods that create MPI windows
n : the number of `store` assignments of integer constants
E : the set of edges between instruction nodes
D : the set of the fact domain

Previously described by subsection 3.6.1, the cost of an IFDS problem solution grows asymptotically towards $O(|E| \cdot |D|^3)$. With $|D| \in O(m \cdot n \cdot 2^b)$ for the property of path sensitivity, the upper bound of the costs is refined as $O(|E| \cdot (m \cdot n \cdot 2^b)^3)$.

When crossing out the costs of the IFDS problem solution, in the preprocessing, there is the Unlock-Map algorithm taking place (subsection 4.1.5) for providing integral information to the IFDS problem solution. For a worst-case scenario that is described by the maximum ratio of back edges by loops to the total number of edges, the cost is bound by $O(|E|^2)$. In a best-case scenario without loops in any of the CFGs, the cost is linear to the number of edges.

With this knowledge, there are two simple tests to do: One test over a constant number of facts but a growing number of instructions, and one over a constant number of instructions but

a growing number of path-sensitive facts. For this reason, the tool has been equipped with a benchmark mode: In a benchmark mode called *B2*, the tool skips any step that is not crucial to the IFDS problem solution functionality: the preflight statistics mode, the post-IFDS problem test suite and the writing of results into the CSV I/O sink.

The test environment takes place on a personal computer composed of an Intel Core i5 4210U and 8GB of RAM, running the tool on a Linux Kernel v4.18, using Phasar v1218 compiled by the Clang C++ compiler v6.0 without optimization flags. Every data point results from averaging the total run time of five runs without outlier elimination, using the *B2* mode.

Test 1: constant facts, growing edges

The hypothesis for the first test is: *The total execution time grows linearly to the number of instructions.* For the setup, a script composes C code files over a growing number of statements that are then compiled into the IR. The code is free of statements that make the number of facts grow (as seen in listing 5.1), and free of loops such that the Unlock-Map algorithm only adds best-case, linear costs.

```
1   struct SomeStruct { bool a; volatile unsigned long b; };
2
3   int main () {
4     // ...
5     SomeStruct *toA = &A, toB = &B;
6
7     // n times repeated code block
8     toA->b = toB->b; toB->b = toA->b;
9     // ...
10  }
```

Listing 5.1: Code to stress the tool over an increasing number of instructions.

The IR code that results from one block is shown by listing 5.2. For a stride size of 10, the test ranges from 10 to 400 statement blocks, representing IR code ranging from 120 to 4800 instructions in addition to a constant offset.

```
1  %13 = load %struct.SomeStruct*, %struct.SomeStruct** %9, align 8
2  %14 = getelementptr inbounds %struct.SomeStruct, %struct.SomeStruct* %13, i32 0, i32 1
3  %15 = load volatile i64, i64* %14, align 8
4  %16 = load %struct.SomeStruct*, %struct.SomeStruct** %8, align 8
5  %17 = getelementptr inbounds %struct.SomeStruct, %struct.SomeStruct* %16, i32 0, i32 1
6  store volatile i64 %15, i64* %17, align 8
7  ; followed by five analogous instructions for the inverse direction
```

Listing 5.2: The IR representation per statement block.

For the interpretation of the results shown by figure 5.1, not only the tool has been subject to the analysis (plot line `mpi_rma_tool`) but also a *none*-run by Phasar (plot line `phasar/none`): The executable of Phasar provides a set of IFDS problem solution plugins for certain simple and recurring problems. By setting the appropriate flag, either the plugin is loaded or none[1].

[1] `$ phasar -D none -m $IR_MODULE_FILE`

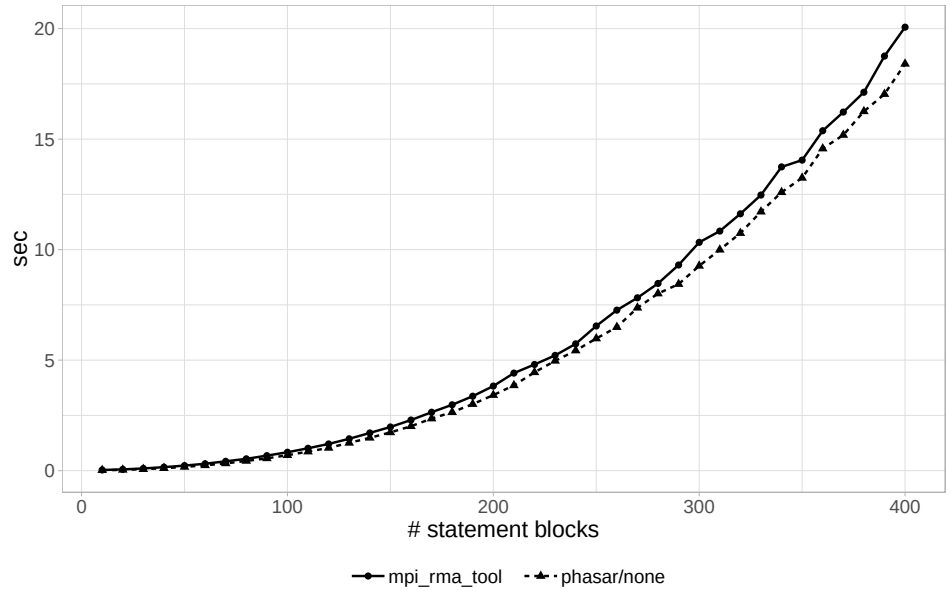


Figure 5.1.: Computational time over a growing program size and constant facts.

Not using any IFDS problem plugin at all gives an idea of the baseline of costs when using Phasar.

In fact, the run time is not growing linearly to a linearly increasing number of instructions: The plot's curves much more resemble a polynomial growth of time, even if no IFDS problem solution is performed. With increasing the program's size, the computation time of the tool performing the IFDS problem solution grows visibly steeper.

The explanation behind this phenomenon lies inside of Phasar: In the preprocessing of the IR code, it performs a *points-to analysis* to handle the function pointer resolution (cf. section 3.5). To perform this analysis, Phasar falls back to the *Andersen's algorithm* [AI94] for which the worst-case run time is bound cubically to the number of instructions of the scoped CFG [SF09].

As the points-to analysis is indispensable to Phasar, even if the costs of a simple IFDS problem solution grow linearly to the size of the program over a fixed set of facts, the run time is already dominated by an algorithm of polynomial computation time and the hypothesis must be rejected.

Test 2: growing facts, constant edges

For this test, it is hypothesized: *For every additional conditional branch the run time doubles.* For a setup that provides programs with an increasing number of path-sensitive facts while keeping the number of instructions roughly the same, again, a script generates C source code files containing a call to `MPI_Win_create`, followed by an increasing number of conditional branches. For each conditional branch, path sensitivity requires every `Win` fact that reaches the instruction to fork into two children facts. The less conditional branches a program instance contains, the more compensation instructions are appended to the end of the program to keep the overall number of instructions at the same level with respect to a maximum N of

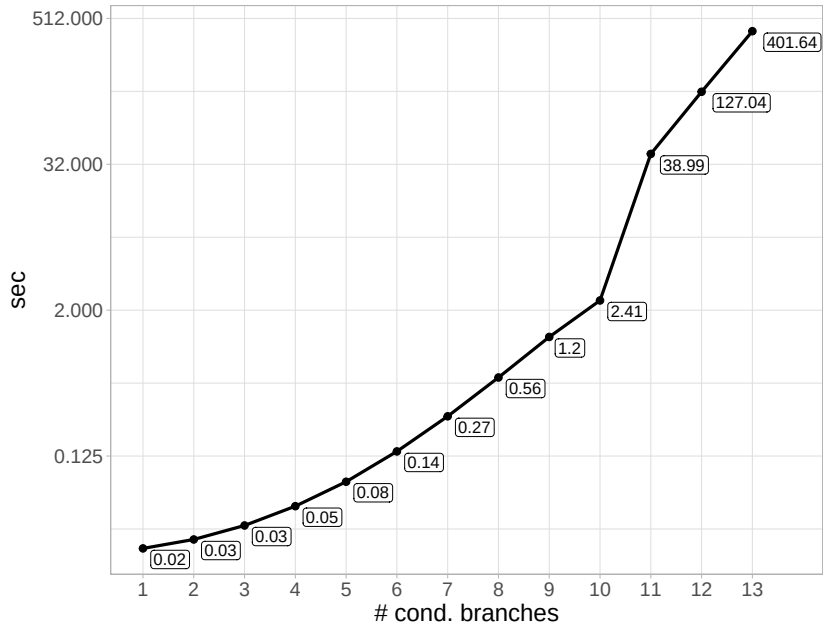


Figure 5.2.: Computation time over a growing number of conditional branches.

conditional branches of the setup.

With $N = 20$, starting with a single conditional branch, every instance of the test programs increases the number by one. The scaffolding of these programs is shown by listing 5.3.

```
1  size_t a_1 = 0, ..., a_20 = 0, b = 0;
2  MPI_Win_create(..., &win);
3
4  // n repetitions of this block, up to N
5  if (1 == a_1) {
6    b += 1;
7  }
8  // N - n compensation instructions
```

Listing 5.3: Code to stress the tool over an increasing number of `Win` facts.

The result of this test is plotted over a logarithmic scale by figure 5.2: Without surprise, the execution time doubles straight up to $n = 10$, then taking a fairly huge jump for $n = 11$ with tripling the time up to $n = 13$, when the test suite had been aborted early due to the unforeseen development.

It is unknown what the precise reason for the impact at $n = 11$ is, possibly a significant loss of cache locality or an increased paging activity by the operating system. With computers more modern and powerful than the one used for the experiments, this effect is probably deferred by one occurrence of a conditional instruction. Further research on this proposition was abandoned in favor of chapter 6.

With the given test hardware, while the hypothesis can be accepted for $n \leqslant 10$, for larger n the performance is currently worse and, with the previous expectation turning into a lower bound, the problem is moved to grow by $\Omega(2^n)$.

Summary

The first test shows that a linear growth of the run time for an IFDS problem solution of Phasar is nowhere in range, even for purposes that do not require path sensitivity. For a worst-case estimation under a constant number of facts, a cubic growth over the number of instructions of the program acts as the upper bound. The second test confirms the assumption of an exponential explosion of the run time over an increasing number of conditional branches inside the program.

For the second test in particular, there is one obvious circumstance: With all the given conditional branches, how many of them actually contribute to the MPI RMA analysis? In this case, they do not but just make the space of states inflate. The next section covers the elimination of such parts of the code.

5.2. Approaches to run time optimization

Again, consider the estimation of the run time development of the IFDS problem solution with the variable definitions of section 5.1: $O(|E| \cdot (m \cdot n \cdot 2^b)^3)$. It is desirable to diminish n, the number of generated `IntConst` facts, and b, the number of conditional branches, down to a level for which they still contribute to the completeness of the analysis of MPI RMA properties while allowing the tool to faster examine programs of non-trivial size. Preferably, any attempt to programmatically reduce these variables largely outperforms the IFDS problem solution algorithm by the required amount of time.

The following, fully optional measurements are proposed:

1. Allow `IntConst` facts to be path-insensitive if no path sensitivity is required. This turns n from a factor into an offset of the complexity.
2. Allow skipping inter-path tests and facts of type `IntConst` altogether if the program is known to not benefit from them, e. g. by the level of rank dynamicity; n can then be eliminated from the complexity completely.
3. Allow scanning and removing of sections of the IR that are conditionally reachable but do not *seemingly* contribute to any static analysis of MPI RMA program code. By cutting off these sections, b and E will shrink.

The following subsections demonstrate the underlying problems and their solutions in the implementation of the tool. Yet, any of the mentioned procedures will remain a threat to completeness under certain conditions, asking for a cautious opt-in by the user.

5.2.1. Beneficial assumptions

By default, facts of type `IntConst` are assumed to require path sensitivity to cover cases as shown by listing 5.4: If the constant assigned to `rootRank` is later overridden by another constant, the change can be detected and tested for the meeting-over-all-paths property for using it in the rank attribution step.

As path sensitivity has been shown to easily inflate the fact domain set exponentially, it

```
1   int rank = -1, rootRank = 1;
2   MPI_Comm_rank(..., &rank);
3
4   if (rank == rootRank) {
5     // ...
6   }
7
8   rootRank = 2;
9   if (rank == rootRank) {
10    // ...
11  }
```

Listing 5.4: Path sensitivity for `IntConst` facts.

overtaxes the program if the pattern shown by listing 5.4 is not being used at all. The user of the tool is given two options to overcome such a situation:

1. Considering all `IntConst` facts immutable: After the first assignment of an integer constant to a location in the IR, no further overwriting is assumed after that, and path sensitivity for this type of facts is suppressed.
2. Skipping the inter-rank analysis altogether: `IntConst` facts currently only serve the purpose to derive assumptions about certain interactions in the context of MPI windows across processes. For a user, there are a couple of reasons to opt-in for skipping these tests: the sheer number of `IntConst` and `Win` facts resulting from the IFDS problem solution, as well as the follow-up costs of the rank attribution and epoch matching algorithms, and also for using communication patterns that are more sophisticated than comparing the current rank against a hard-coded constant.

For the implementation of the tool, a user can set the respective flags for either case: If the assumption of immutable rank roles is safe, it favors the run time for taking this consideration into account, and if there is no need to perform any of the mentioned tasks, the user is rewarded by saving precious time.

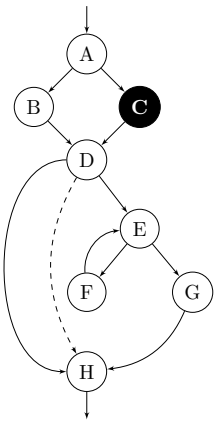
5.2.2. Graph slicing

Considering the *basic block*s (BBs) of a CFG of a given procedure, the set of instructions of some BB perhaps does not contribute anything to the static analysis of MPI RMA, let alone have any kind of calls to library methods at all. While this concern alone is not the biggest cost driver, path sensitivity over these seemingly meaningless sections of the CFG is a major issue. If the tool can eliminate certain seemingly irrelevant sections of the CFG at reasonable costs in advance, the overall run time may greatly benefit.

First, the precise criteria for the relevance of a BB are given:

1. There is a call to a declared method that is subject to the IFDS problem of MPI RMA: setup of MPI communicators, windows, remote actions, synchronization, and some auxiliary functions, e. g. `MPI_Comm_rank`.
2. Unless disabled by measurements of subsection 5.2.1, there is a scalar assignment of an integer constant to a memory location.
3. There is a call to some other defined procedure of the given IR module.

```
1  // A
2  if (someCond) {
3    // B
4  } else {
5    MPI_Barrier(MPI_COMM_WORLD); // C
6  }
7
8  // D
9  if (someOtherCond) {
10   // E
11   while (yetAnotherCond) {
12     // F
13   }
14   // G
15 }
16 // H
```



Listing 5.5: Source code (left) and its colored CFG (right).

In this chapter, a *seemingly irrelevant* block refers to a BB that does not meet any of the listed criteria. While actions relevant to MPI RMA can be widely identified over their distinct method names, there are some notable exceptions that drain completeness: buffer actions, local actions, copying and aliasing of MPI handles, and remote actions by accessing locally mapped memory of remote processes as it is the case for the shared window flavor. In a simplified scan over the instructions of a BB, these actions cannot be identified for having MPI RMA semantics, thus any such BB is only seemingly irrelevant and may be wrongly erased from the CFG, possibly leading to an invalid IR as described on p. 83, making this optimization approach fully optional.

In the context of this thesis, a *flow net* refers to a directed, unweighted graph with nodes of which exactly one is an outlined start or *source* node and exactly one is a *sink* node – a node without any outgoing edge. The source and sink nodes must be distinct. In this chapter, a CFG is again composed of nodes representing BBs, unlike the IFDS problem instance of chapter 4 that treats single IR instructions as nodes. Every node is given a *color*: *white* for being seemingly irrelevant, *black* otherwise. With loss of generalizability, for a CFG, a *sub-flow net* refers to a subgraph of that CFG that is a flow net, and that consists of a *dominator* node, a start node that is branching, and a *zip* node, a source node that is merging.

Wormholes

The first idea of graph slicing is to find all largest – given the number of nodes – sub-flow nets of a CFG with every non-dominator and non-zip node being white, and to replace every finding by a single edge, the *wormhole*.

Listing 5.5 shows an example of an excerpt of source code and its CFG. The node C is given the color black for containing a call to `MPI_Barrier`. Thus, the subgraph induced by the nodes $\{A, B, C, D\}$ cannot satisfy the definition of a disposable sub-flow net. Taking the induced subgraph of the nodes $\{D, E, F, G, H\}$, there is a dominator node D and a zip node H: In this case, a wormhole can be introduced by removing the branching property of D and inserting a

single outgoing edge (D, H) while removing the nodes $\{E, F, G\}$, saving two points of forking.

The algorithm for cutting off sub-flow nets to replace them by wormholes operates on each `llvm::Function` of the given `llvm::Module`, starting by performing a *depth-first search* (DFS) over the `llvm::BasicBlocks`: On visiting each node, an overlay structure resembling the inverted CFG over the BBs is created, as well as the color and additional attributes are assigned.

A second DFS finds dominator nodes, possibly starting from a virtual super-start node of the inverted CFG and reaching out to the original start node, the sink-node of the inverted CFG. A dominator is given under one of the following conditions:

- From the perspective of a regular CFG, it is the top-most branching node,
- it is any other branching node if no previous dominator is known at that point ($\perp$) or
- it is a branching node that is also the zip node of some sub-flow net graph.

When reaching a branching node b on the first visit, the DFS returns to the parent, taking a symbol b_r and the dominator of b, or b as the dominator if there is no previous one set, together with other symbols of every other first visit. By touching b over the second incoming edge, it returns a symbol b_l as well as the dominator information. Any branching node b is considered *resolved* if at a merging node, after returning from all children, b_l and b_r were collected for mutual annihilation; unresolved symbols are returned to the parent node.

Dominance is thrown over under one of the following conditions:

- The color of a node n is black: If it is not a branching node at the same time, the dominator is set to $\perp$; otherwise, n becomes the new dominator.
- After returning from all children, a merging node m observes conflicting information about the dominance of its children. If m is not a branching node at the same time, the dominator is set to $\perp$; otherwise, m becomes the new dominator.

Whenever dominance is overthrown, the set of unresolved symbols is cleared as well, allowing the detection of a smaller sub-flow net. A largest sub-flow net without any black intermediate nodes between some node d and z is found if both of these conditions hold:

- At a merging zip node z, after resolving the symbols from all children, there are no unresolved symbols left. For every branch node b starting from d, both outgoing edges and their symbols b_l, b_r have been resolved somewhere no later than z.
- All children of z report d as the observed dominator: the meet-over-all-paths dominance property.

When treating the CFG of listing 5.5 as a complete graph, the procedure to determine the sub-flow net is visualized by figure 5.3. Loops, such as the `while`-loop around the nodes $\{E, F, G\}$ require additional attention: If E was treated as a conditional branch block, it could not answer the question of dominance when touched by the edge (F, E) as it still awaits its return from F to determine this answer. Thus, on discovery of a node that is still pending, the complete branch is grayed out (indicated by the γ labels) to treat the evaluating loop component as a non-conditional branching node, never emitting symbols and never qualifying for dominance.

Internally, the symbol resolution is implemented using a symmetrical set difference operation,

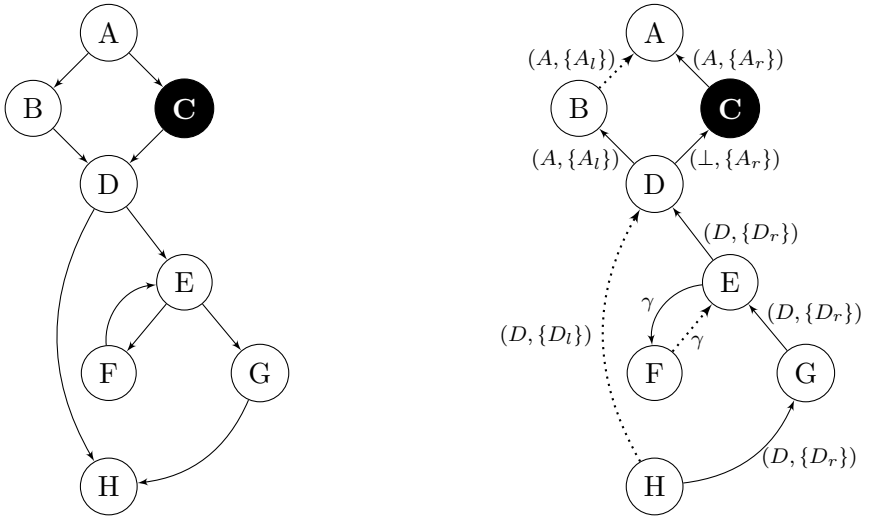


Figure 5.3.: CFG (left), and inverted after the second DFS (right).

with its performance costs growing linearly to the size of the set of symbols to operate on[2]. For a merge node having m incoming edges, the operation takes place $m-1$ times. For N nodes of all of the CFGs, the overall number of incoming edges of merge nodes is bounded by $O(|N|)$. With a complexity of the DFS in $O(|N| + |E|)$, the growth of the overall costs of this procedure is coarsely bounded by $O(|N|^2)$.

Compression of conditions

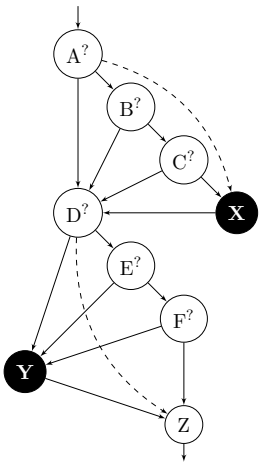
For a sequence of Boolean expressions, the evaluation of each single one results in a BB with conditional branching. When writing software, it is common to line up these expressions conjunctively or disjunctively, for example inside of an `if` or `while` statement. If there is a black BB reached by the *all-true* case of a conjunctive set of expressions, or by the *at-least-one-true* case of a disjunctive set of expressions, and every evaluating BB is white, this part of the CFG can be compressed into two edges, with one reaching the black BB while its counterpart does not.

Consider the example of listing 5.6: With two calls to MPI RMA methods, both of them are guarded by an expression, first by a conjunction of conditions, then by a disjunction. For maintaining path sensitivity, the node set $\{B, C, E, F\}$ increases the number of generated facts. For the implementation of the static analysis of MPI RMA, the false-case of the nodes $\{A, B, C\}$ is seemingly equivalent to a single edge (A, D), as well as their the true-case is seemingly equivalent to a single edge (A, X). Analogously, these assumptions are made for the disjunctive test encoded by the node set $\{D, E, F\}$. If the CFG can be compressed by skipping the intermediate BBs as indicated by the dashed line, a single fact reaching the node A will only generate four new facts until leaving Z – instead of 16.

The implementation reuses the ideas of the search for wormholes: A first DFS creates an

[2] `std::set_symmetric_difference` on `cppreference.com` – `https://en.cppreference.com/w/cpp/algorithm/set_symmetric_difference`, last seen 2018-12-29.

```
1  if (condA && condB && condC) {
2    MPI_Get(...); // X
3  }
4
5  if (condD || condE || condF) {
6    MPI_Put(...); // Y
7  }
8  // Z
```



Listing 5.6: Source code (left) and its colored CFG (right).

inverted, colored CFG. Instead of looking for a largest sub-flow net, any sufficiently large one is taken under the following conditions:

1. The zip node z has three or more incoming edges (conjunction) or two incoming edges with the black node having more than one incoming edge (disjunction),
2. all children are derived from one dominator node d and
3. of all these children, exactly one is a black node that is not itself the dominator.

Unlike the search for wormholes, there is no need to return edge symbols on every branching node for resolution. Instead, fixed-size returns are sufficient: the dominating branch node and the edge of descendance, the last black node if available, and a flag to indicate previous merging at a child of first degree. Non-branching, white nodes act as a pass-through of return information.

Dominance is reset or claimed or both on one of the following conditions:

1. At a branching node, it is reset if the node acts as an evaluation node of a loop,
2. at a branching node when no or no consistent dominance is currently known – or it just had been reset by the conditions below,
3. at a zip node, allowing the dominator to jump directly to the black node (conjunction) or to the zip node (disjunction) over one of its edges,
4. at a zip node, if there is more than one child of black color or
5. on the occurrence of a black node that is in the line of descendance of another black node.

Figure 5.4 shows how to apply the DFS under the given specification to the CFG of listing 5.6 when considered complete. Let $R = \{l, r\}$ denote the set of the sides where the DFS returned from a dominator, $H = \{\top, \bot\}$ the Boolean flag indicating precedent merging of a non-dominating node, then the edges are labeled by $N \times R \times H$, not showing the last known black node in the returned tuple.

Detecting the sub-flow nets representing a conjunction or disjunction as specified by this

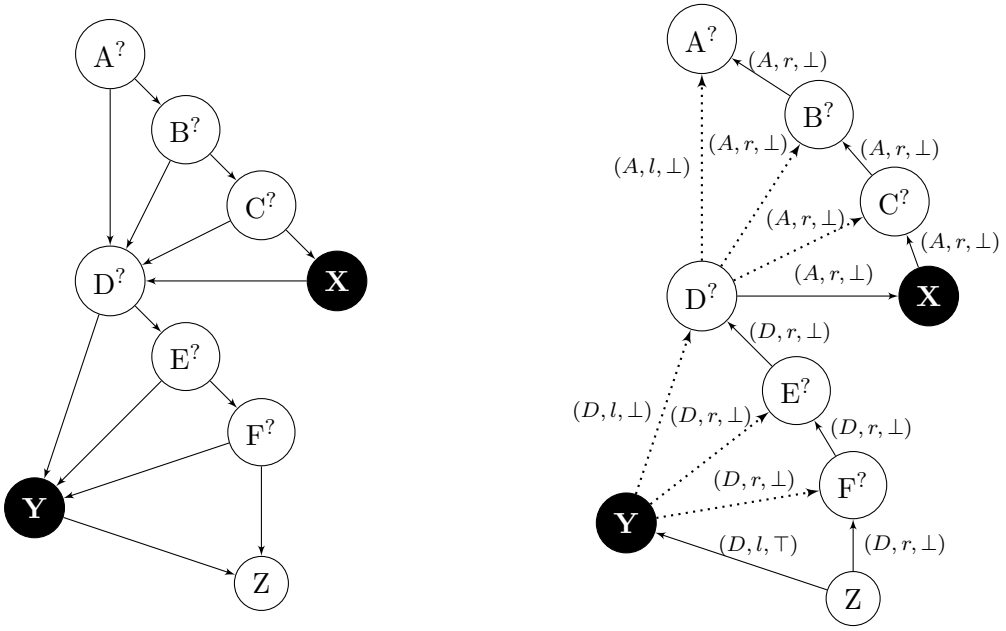


Figure 5.4.: The colored CFG (left), and after the DFS (right).

subsection, as well as replacing one edge of the dominator by an edge jumping over seemingly irrelevant branch nodes can be done inside of a regular DFS. Thus, the cost of this procedure grows no worse than the one imposed by a DFS: $O(|N| + |E|)$.

5.2.3. Window isolation

In an attempt to allocate the available computer resources more granularly, the implementation supports *window isolation*: In any MPI RMA program that makes use of more than one window, it may be useful to restrict the IFDS problem solution to a single window at a time, setting $m = 1$ in the growth estimation of section 5.1. The IFDS problem model keeps track on the occurrences of window setup calls globally, allowing the generation of a `Win` fact on the i-th attempt only. Additionally, this concept allows a simple yet effective way to parallelize the static analysis of a program by its number of window generation calls, free of requirements to synchronize any computation nodes, but sacrificing multiple tests of subsection 4.2.3 and subsection 4.2.4 with respect to completeness.

5.2.4. Estimation and progress

With the scalability analyzed by section 5.1, and the various optimization approaches presented by the previous subsections, it is worth estimating the dimension of the underlying IR module with respect to the IFDS problem solution in advance to schedule computational resources, to pick optimizations, or to reduce the size of the unit under analysis. Two features have been implemented for assisting the decision: a *preflight* and a progress display.

Preflight

The preflight serves to count the following elements of the IR module:

1. The overall number of instructions that a zero fact will encounter,
2. the offset of instructions to the generation of the first path-sensitive fact with respect to the settings of subsection 5.2.1,
3. the number of calls to methods that generate `Win` facts,
4. the number of calls to methods of the MPI RMA domain,
5. unless `IntConst` facts have been disabled by the user, the number of LLVM `store` instructions writing an integer constant, with a tracing feature to skip counting of overwrites and
6. the number of conditional branch instructions.

The implementation of the preflight is done by a single-fact IFDS problem model to be solved by Phasar. The result set of the preflight provides two advantages in a comparably reasonable amount of time – polynomial, as shown by the first test of section 5.1:

1. Acquiring numbers for pessimistically calculating the upper bounds of the number of facts and instructions that the main IFDS problem solution will face and
2. measuring the impact of the optional optimizations that have been presented in this chapter as the preflight takes place after slicing the CFGs of the module.

Progress

When running the MPI RMA static analysis on Linux or any platform that supports POSIX signals, and if the preflight grants the assumption of the total number of `Win` facts fitting into an unsigned 64-bit counter, then a user may optionally enable the display of progress at the default output channel: Using `SIGUSR` signals, an update is given for every generation, fork and killing of a `Win` fact. The display not only enables an estimation of the overall progress of the IFDS problem solution with respect to the upper bound of `Win` facts, but also to roughly measure the time span between launching the program and the first touch of a `Win` fact for the many operations occurring in the mean time (cf. subsection 4.1.1 and section 5.1).

5.2.5. Overview

Table 5.1 lists all steps of this section for optimizing the IFDS problem solution run time in their order of execution. The column *Complexity* describes the growth of costs that is added to the respective *Stage* of the tool; the degree of amortization varies as shown by the next section, depending on the underlying source code.

Optimization / Estimation	Stage	Complexity
graph slicing: wormholes	preprocessing	$O(\|N\|^2)$
graph slicing: compressing conditions	preprocessing	$O(\|N\| + \|E\|)$
preflight	pre-IFDS problem	$O(\|E\|)$
immutable `Int` facts	IFDS problem	$O(1)$
inhibited `Int` facts	IFDS problem	$O(1)$
window isolation	IFDS problem	$O(1)$

Table 5.1.: Costs of optimization and estimation techniques.

5.3. Examples

After considerations of MPI communication, LLVM, the static analysis of MPI RMA properties over an IFDS problem and follow-up models, and approaches to quench concerns related to scalability, the implementation is at a state to face the confrontation of non-trivial programs. In contrast to non-RMA MPI programs, the current availability of client programs using RMA at a reasonable size that suit reproducible setups is comparably scarce, possibly reflecting the overall adaption of MPI RMA at large[3]. Nonetheless, two suitable collections of source code are presented in this section.

5.3.1. OSU Micro-Benchmarks

Hosted by *The Ohio State University* (OSU), its MPI implementation is known as *MVAPICH* [WH06]. Alongside the implementation, a collection of C and C++ source code files known as *OSU Micro-Benchmarks* (OMB) is provided to serve as a code base to perform benchmarks of different aspects of an MPI implementation, such as RMA.

The programs to benchmark MPI RMA are named: `osu_put_latency`, `osu_get_latency`, `osu_put_bw`, `osu_get_bw`, `osu_put_bibw`, `osu_acc_latency`, `osu_cas_latency`, `osu_fop_latency` and `osu_get_acc_latency`. The ones of the suffix `_latency` represent latency benchmarks, as `_bw` does for bandwidth and `_bibw` for bidirectional bandwidth. It is common for all of these programs to offer flags to the user to control the window flavor and synchronization mode at launch time. Thus, in most of the source code files, every flavor and every mode is encoded by default. As the OMB suffer from an early aliasing of the `MPI_Win` handle to use, slightly modified versions have been used for testing, replacing the call-by-reference of the setup procedures by a return of the newly assigned `MPI_Win` identifier.

In the first step, every program's C source code file is compiled together with its dependencies into the LLVM IR using Clang v6.0 with optimizations disabled. In order to obtain an impression of feasibility of a static analysis, a preflight-only run is taken for each of the programs with different degrees of run time optimization[4].

[3] As of 2019-01-01, on StackOverflow, there are 23 topics labeled by `mpi-rma`, in contrast to 5068 topics labeled by `mpi`. Counted by `https://stackoverflow.com/tags`.

[4] `-OW`: Attempt to find wormholes and reduce the CFGs. `-OR`: Skip the inter-rank analysis completely.

The results are shown by table 5.2. While none of the programs is particularly big or small regarding the number of IR instructions, the most aggressive optimization options are shown to be notably effective, reducing the conditional branch instructions for `osu_cas_latency` by more than 20%, hence the potential number of `Win` facts by up to factor 2^{82}. Unfortunately, even for the smallest program (`osu_put_bibw`) with all possible optimizations, an upper bound of 229 conditional branches with four window setups is expected to remain far beyond feasibility for the static analysis.

Program	# win.	Flags	# instr.	# cond. branches
osu_acc_latency	4		3872	343
		-O1	3600	315
		-O2 -O3	3514	303
osu_cas_latency	4		3882	337
		-O1	3602	307
		-O2 -O3	3094	255
osu_fop_latency	4		3853	337
		-O1	3573	307
		-O2 -O3	3287	275
osu_get_acc_latency	3		3376	299
		-O1	3247	289
		-O2 -O3	3161	277
osu_get_latency	4		3902	354
		-O1	3630	317
		-O2 -O3	3544	305
osu_put_latency	4		3864	343
		-O1	3592	315
		-O2 -O3	3506	303
osu_get_bw	4		3712	316
		-O1	3433	288
		-O2 -O3	3347	276
osu_put_bw	4		3851	323
		-O1	3572	295
		-O2 -O3	3486	283
osu_put_bibw	4		3084	272
		-O1	2793	242
		-O2 -O3	2704	229

Table 5.2.: OMB MPI RMA preflight data.

5.3.2. TrackerSim

Maintained by the observatory of the *University of Hamburg*, *PHOENIX* is a software to perform simulations in the context of astronomy while also being a long-term user of the MPI environment [BH98][AWH16].

TrackerSim is an attempt to mime certain sections of the PHOENIX source code that originally use OpenMP for local parallelization by an implementation using MPI RMA instead [Squ16]. Unlike the OMB, TrackerSim is written in Fortran, thus requiring the Flang compiler when requesting an LLVM IR representation. TrackerSim consists of multiple MPI RMA variants:

- *Fence* (`ts_fence`). Using statically allocated windows and the fence synchronization mode.
- *Lock* (`ts_lock`). Using statically allocated windows and PTS.
- *Shared* (`ts_shared`). Using the shared window flavor and PTS.
- *SharedNoRMA* (`ts_shared_inner`, `ts_shared_outer`). Two variants using the shared window flavor, PTS and implicit communication, distinguished by the placement of MPI RMA locks.

All variants are compiled by the Flang binary distribution 2018-09-21 with optimizations disabled as well. Again, the first step is the preflight for every variant and degree of optimization.

The preflight results are shown by table 5.3. By the number of instructions, at a first glance,

TrackerSim is on a par with OMB. Nonetheless, it turns out to be mostly immune towards attempts to reduce its CFG. The inspection of the IR files revealed interesting properties in comparison to Clang:

- Module-wide `INTEGER` variables have been merged into a larger contiguous array, not having individual symbols.
- On demand, this array is addressed by an offset, followed by an `llvm::BitCastInst` instruction that models the cast of an array cell into a scalar integer value.
- This procedure is repetitive across the file, not doing any reuse of casted symbols.

As a consequence, IR files by Flang on the `-O0`-optimization level yield an impression of cluttering. Both for attempting to reduce the number of conditional branches, and for reducing the overall number of instructions as well, the two smallest variants of TrackerSim (`ts_fence`, `ts_lock`) were recompiled using the `-Os` flag, requesting the compiler to optimize the compilation for a reduced size of the executed binary file.

Program	# win.	Flags	# instr.	# cond. branches
ts_fence	5		2682	61
		-OW	2682	61
		-OR -OW	2645	60
ts_lock	5		2595	57
		-OW	2595	57
		-OR -OW	2558	56
ts_shared	5		3773	74
		-OW	3768	73
		-OR -OW	3731	72
ts_shared_inner	5		3970	75
		-OW	3965	74
		-OR -OW	3928	73
ts_shared_outer	5		3960	75
		-OW	3955	74
		-OR -OW	3918	73
ts_fence_Os	5		1158	50
		-OW	1155	49
		-OR -OW	1143	48
ts_lock_Os	5		1124	47
		-OW	1121	46
		-OR -OW	1109	45

Table 5.3.: TrackerSim preflight data, with `-Os` programs.

As demonstrated by table 5.3, the impact of the compiler option is rather huge from the perspective of the preflight, reducing the total number of instructions by more than 50%[5] and the number of conditional branches by a notable degree, leaving attempts of shrinking the CFGs again at a petty level. Again, under the observation of section 5.1, even the smallest program (`ts_lock_Os`) remains infeasible to analyze on a personal computer.

5.4. Summary

In this chapter, a number of notable observations have been made with respect to the feasibility of a static analysis with Phasar and the model implementation of an MPI RMA static analysis in particular:

- By default, when using Phasar, there is no linear growth of costs over an increasing program size but a polynomial one.
- Path sensitivity as an IFDS problem contributes exponentially to the growth of the

[5] Actually, this is of a cheating nature: Flang largely achieves this by inlining `llvm::BitCastInst` instructions, a legal way by the IR specification. Regardless of this method, Phasar only perceives the subset of instructions, resulting in a smaller number of edges.

run time: The feasibility currently reaches its pain threshold slightly above of a trivial program complexity, having between 10 and 13 conditional branches.

- Attempts to reduce a CFG by removing seemingly irrelevant parts and skipping the inter-rank analysis show ambiguous results: Satisfying ones in case of the OMB programs, neglectable ones for TrackerSim. It is currently unknown whether this is a matter of the actual source code or the two different compilers.

None of the tests was performed successfully on any example program as the test platform always ended up in memory excess during the IFDS problem solution after roughly forty minutes. Overall ideas and suggestions, especially to the show-stopping performance of the current implementation, are given by the next two chapters.

6. Remodeling the IFDS problem

Given the results of chapter 5, further refinements of the IFDS problem design are discussed, presented and given a re-evaluation.

Considering the perspective of the scalability measurements of section 5.1 and the preflight data of the programs under analysis in section 5.3 as discouraging, it is worth putting effort into modeling the IFDS problem in a way that circumvents path sensitivity a priori, avoiding an exponential growth of the state space as an average expectation over the determination by the number of conditional branches.

While the first approach of the IFDS problem model allows an arbitrary *fat state* of facts, the overall idea now is to restrict the generation of facts to the traversal through production rules of the context-free grammar specified by section 4.2 while also allowing the reuse of previous achievements of the pre-IFDS problem stage, such as the Unlock-Map algorithm (subsection 4.1.5) and the graph slicing (subsection 5.2.2), but also by using the first IFDS problem model as a reference in the testing process.

6.1. The revised model

In contrast to the first IFDS problem model of subsection 4.1.3, the range of fact types is trimmed to `Null`, `Comm` and `Win`. In addition, the following simplifications are made to the implementation:

- Symbol tracing is limited to the tracing of the own handle,
- inter-fact referencing is restricted to the direct ancestor relation between `Win` facts,
- no information on post-IFDS problem aspects is collected along the way, such as *true* or *false* branching and calls to the rank value acquisition,
- the history of `Win` facts is strictly limited to verbose calls of methods that contribute to the grammar and
- as a result of the previous circumstances, tracking of the implicit communication in the shared window flavor is not supported by the implementation.

The notable change to the `Win` fact type lies in the behavior of creating successor facts: Previously, the fact domain exploded by a sweeping forking on reaching a conditional branch instruction, regardless if a fork was considered promising with respect to the upcoming nodes for both successors – at best, such a region of the CFG could have been replaced by a wormhole (cf. subsection 5.2.2) for not contributing to the static analysis.

In the second IFDS problem model, calling it the *lazy mode* of the initially presented IFDS problem, window facts no longer fork on crossing a conditional branch: Instead, each

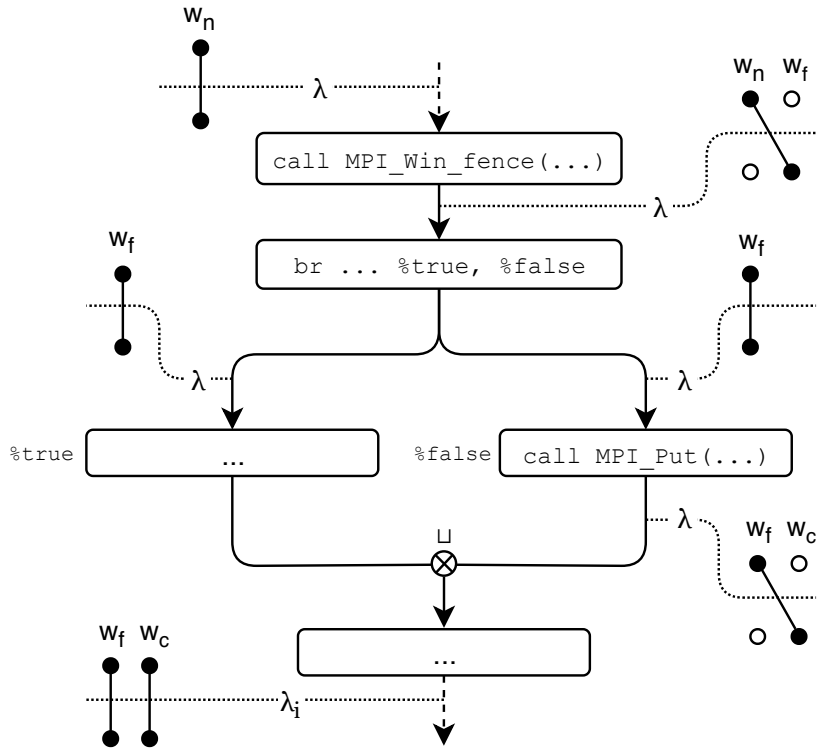


Figure 6.1.: Transition-driven flow mapping of a window state.

window fact is generated for living no longer than the single state it is representing between two occurrences of MPI RMA calls, or one call and the end of the CFG. This way, two paths following a conditional branch will not make the fact space grow unless one or both of them change the window's state on either path. If an upcoming call to an MPI RMA method is covered by the grammar specification of section 4.2, the incoming `Win` fact object generates a successor that attempts to advance over the legality of the transition and continues in the graph reachability analysis in case of success, leaving the parent killed.

This behavior is demonstrated by figure 6.1, taking a CFG derived from the one of figure 4.3 (p. 40): A newly created window object is denoted w_n, and upon reaching the call to `MPI_Win_fence`, its successor w_f now represents the fenced-state. As w_f is distributed across both branches, it is killed by a call to `MPI_Put` in the `%false`-node and replaced by w_c, the window in the state of non-synchronized fence-communication. At the flow function λ_i, Phasar will provide the two facts w_f and w_c: If the call to `MPI_Put` had been absent, at this point, there would have been w_f only.

Another computationally significant change has been made to the handling of loops: Instead of stabilizing `Win` facts by locking the individual facts over the information by the unlock map on facing the re-entry of a loop (cf. subsection 4.1.5), the respective flow functions can now perform an early elimination on the edge that leads into the loop's body for facts that have been newly generated inside of that particular body, saving a large amount of otherwise dry processing of locked facts as part of the IFDS problem reachability analysis.

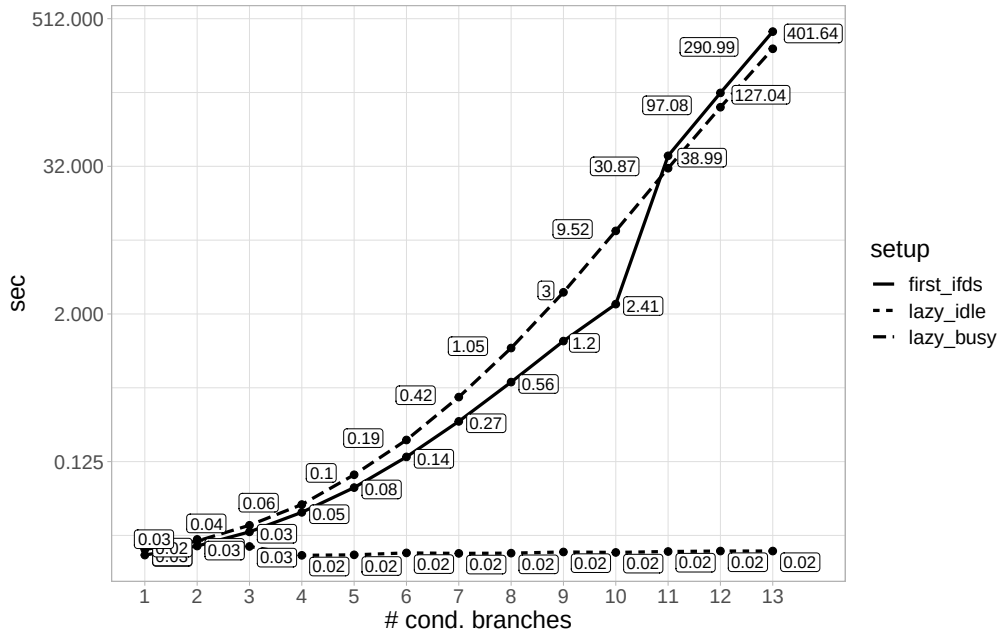


Figure 6.2.: Computational time over a growing program size and constant facts.

6.2. Scalability

The lazy mode is unable to computationally perform better on programs that actually model an exponential growth of the `Win` state space, yet it is expected to outperform in the average case: non-artificially crafted source code. This assumption is derived from repeating the test 2 of section 5.1:

In the previous test setup, there was a customizable number of conditional branches that lead to code presumably irrelevant to the static analysis of MPI RMA, calling this the *idle* setup. Now, in a second version of this test, the code inside of the conditional branches is replaced by calls to an MPI RMA method, making this test the *busy* one.

For the revised model, the idle setup is expected to have a stable and insignificant low consumption of time over an increasing number of conditional branches whereas the cost of time of the busy setup will remain at growing exponentially – possibly at a lower scale due to the overall lighter implementation. Indeed, these assumptions are confirmed by the measurements depicted by figure 6.2.

As the idle setup will never generate more than a single `Win` fact under the lazy mode of the IFDS problem (plot line `lazy_idle`) its run time remains constant. It is unknown what exactly causes the busy setup's run time to initially grow steeper without hitting the step as observed in the first model (plot lines `lazy_busy`, `first_ifds`) at $n = 11$ but at the scale of the test setup, given 13 conditional branches, the time to finish is reduced by nearly two minutes.

Estimating the actual number of different window state transitions of TrackerSim between 100 and 1000 per window, the IFDS problem solution in the lazy mode is expected to pivot around ten seconds for one given window on the test hardware – a fairly promising duration.

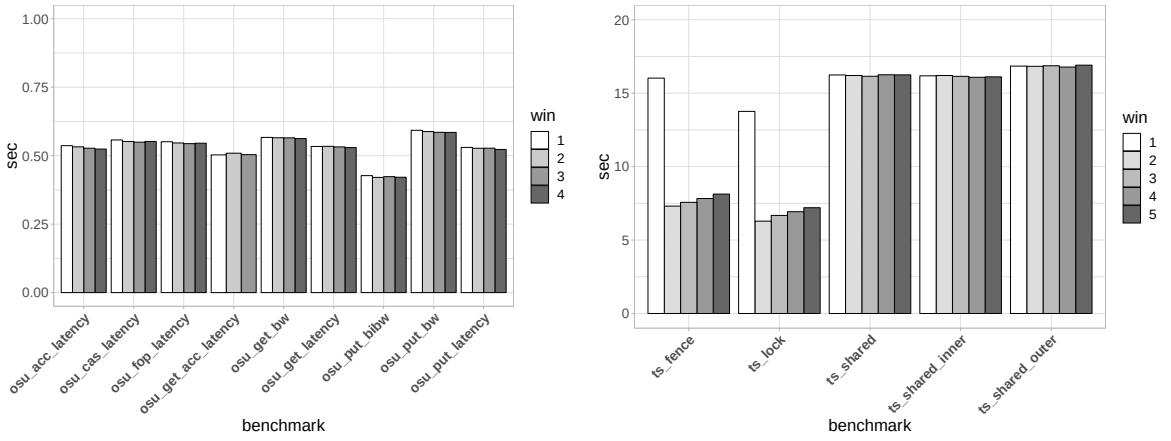


Figure 6.3.: OMB (left), TrackerSim (right).

6.3. Examples

Again, the examples under analysis are TrackerSim and the *OSU Micro-Benchmarks* (OMB). Ranging from three to five window instances per benchmark, every i-th window has been analyzed by the window isolation mode (cf. subsection 5.2.3), again averaging the run time over five runs.

The resulting run times, grouped by the programs and window instances, are presented by figure 6.3. For the OMB, the required run times are homogeneously located around the half of a second. In case of TrackerSim, more dynamics can be observed: Both the shared and non-shared instances significantly differ by the required time, partially explained by the up to 50% higher number of lines of IR code of the shared-flavor programs, as well as having peaking efforts in analyzing a single window in the non-shared programs, code-wise for roughly twice the amount of actions taking place on the CFG for the first windows.

In addition, for none of the presented programs the lazy mode IFDS problem solution reported any violations under the grammatical specification.

6.4. Summary

Considering a different approach to model the IFDS problem for testing an essential MPI RMA property has been shown to be a success with respect to scalability under ordinary circumstances: For the presented examples, despite their overall potential size of the state space, the static analysis has turned into a matter of seconds, even on a personal computer.

Nonetheless, the lazy mode IFDS problem model currently sacrifices any tests beyond the initial IFDS problem solution: From the lack of the ability to properly order and group `Win` facts afterwards to the missing of `IntConst` facts that may be useful to further investigate issues across paths and even MPI ranks. Yet, the model may serve as a basis for further enhancements to the overall IFDS problem specification towards recovering features from the initial, naive attempt to model path sensitivity and inter-fact relations.

7. Related work

Other work in the context of static analysis and MPI – and more globally: APIs – is reported by this chapter.

With the topics of static analysis, IFDS problems, LLVM and MPI being a wide field for related work, this chapter focuses on publications that closely cover the core topic of this thesis: the typestate analysis of API objects. The individual summaries attempt to highlight concepts of the problem modeling and approaches to tame the state space explosion.

MPI & Clang

In a close context, A. Droste et al. have already implemented a static analyzer of certain generic properties of MPI by using Clang [DKL15]. A version of this tool is now part of the official Clang repository providing path-sensitive and syntactical tests. The path-sensitive set includes tests for mismatches in pairs of non-blocking MPI method calls and the respective call to a `Wait`-method; the syntactic analyses identify type incompatibilities between types of the programming language and MPI, as well as bad memory references when using MPI methods. Although being limited to the family of the C languages, the tool has been shown to detect violations in open source projects using MPI.

The tool by Droste demonstrates how Clang serves as a framework for static analysis, allowing a developer to hook into multiple stages and tools around the Clang environment:

- *Diagnostic.* A diagnostic is best described as a compiler feedback being tightly bound to features of the programming language in use, even to its surrounding macros. Aside from reporting errors, maintaining genericness over any kind of use case or target platform, it is known to developers for the wide set of `-W` flags[1] in the driver invocation of both Clang and compilers by GCC.
- *Clang-tidy.* `clang-tidy` is a stand-alone tool providing a wider range of static code analysis tests: Given the *abstract syntax tree* (AST) of a program, a *check* defines AST-based matchers, testing any match reported by the tool for properties. Currently, `clang-tidy` fits multiple uses[2]: domain-specific, local checks (e. g. for *Android*, *Boost* or MPI) and the analysis of both specific (e. g. conventions by *Google* or LLVM) and generic coding styles and conventions (addressing performance, portability or readability), also known as a linter program.

[1] Clang 8 documentation – Diagnostic flags in Clang: `https://clang.llvm.org/docs/DiagnosticsReference.html`, last seen 2019-01-12.

[2] Extra Clang Tools 8 documentation – Clang-Tidy: `http://clang.llvm.org/extra/clang-tidy/`, last seen 2019-01-12

- *Clang Static Analyzer* (CSA). The CSA is another stand-alone program, providing an IDE integration and visually enriched results of path-sensitive static analyses, allowing the user to trace violations across the source code. Again, there exist multiple *checkers*[3] examining the source code for generic (e. g. null pointer dereferencing) or specific violations (e. g. MPI, C library or POSIX methods), given a path-sensitive context.

FLAVERS

G. Naumovich et al. presented a tool named *FLAVERS/Java* for testing concurrency properties of programs using the native threading API of Java [NAC99]. It makes use of concepts that are similar to this thesis: In a first attempt, CFGs are attempted to be shrunk over the absence of user-defined events at a node. A so-called *trace flow graph* is constructed from inlining all CFGs, with a layer of start and end nodes of each thread mapping to the underlying CFG. An additional set of nodes and edges models synchronization and concurrency between execution paths. Also, a FSM accepts a language of valid sequences of API method calls. In a *domain specific language* (DSL), it allows the specification of feasible thread interactions for synchronization models such as locks or monitors. The original *FLAVERS* was designed for the Ada language, and although its inter-thread interactions have been subject to research for improving the analysis performance, there is no explicit information on measurements to reduce the intra-thread state space with respect to path sensitivity [NCC99].

Microsoft SLAM

A project by Microsoft named *SLAM* targets the static analysis of C source code in order to test drivers for Windows API compliance [BR02]. In a first step, the C program code is translated into a *Boolean program* that maintains the original CFG but exclusively uses Boolean variables to model predicates. In the next step, it is tested whether an erroneous state is reachable over some path, otherwise satisfying the property specified by a DSL named *SLIC*. In case of a violation in the Boolean abstraction, the feasibility of a violating path is tested for the original program, possibly ending up in undecidability and demanding human assistance. In another step, the Boolean program is refined by the insights of the previous step, modifying the coarseness of predicates, and then revisited. Similar to this thesis, the reachability analysis suffers from an exponential growth of the number of program states in a worst-case scenario. The authors claim to have reached a fair degree of scalability by carefully adjusting the precision of predicates in the Boolean abstraction.

Object aliasing & permissions

The work of K. Bierhoff and J. Aldrich addresses the topic of aliases of objects, having this kind of challenge in the context of verification of protocol compliance of Java APIs in mind [BA07]. With respect to aliasing of integers and to MPI, that uses opaque and C-compatible handle types preventing rich object-oriented semantics as provided by C++, this topic is of less relevance to this thesis in particular, yet it demonstrates the issue of *permissions* when

[3] Clang Static Analyzer – Available Checkers: `https://clang-analyzer.llvm.org/available_checks.html`, last seen 2019-01-12.

attempting to verify API usage in an object-oriented environment. For Java, Bierhoff and Aldrich have developed a tool called *Plural* out of this research, allowing users to enrich certain types of object-based APIs by annotations for automated verification purposes [BA08].

Tracematches as an IFDS problem

N. Naeem and O. Lhotak cover the modeling of interactions of objects in a typestate analysis using a generalized IFDS problem, again demonstrating the application to Java code [NL08]. With a formalism called *tracematches*, that allows the specification of labeling object interactions, and using these labels to compose a regular expression of the accepted language of interactions, the authors propose to use two IFDS problem solution passes: one for the objects, and a following one for the tracematch analysis. Worth being mentioned is the idea to model the tracking of transitions towards the tracematch state over an IDE problem model.

IDE problems

An *inter-procedural, distributive environment problem* (IDE problem) is a generalization of an IFDS problem, formalized by the inventors of the IFDS problems, M. Sagiv et al. [SRH96]: While an IFDS problem is primarily seen as a reachability problem, allowing the context-sensitive mapping of the liveliness state of a fact to a node of the CFG, an IDE problem formally models the piggybacking of values obtained along the path of a fact, with the join of two or more values states depictable as a *lattice*. Thus, a join operation of a fact over distinct value states may increase its ambiguity, for example in an attempt to model a propagation of constants. In a formal IFDS problem, the value state is simply one of the two states *present* or *not present*. Next to the solver of IFDS problems, Phasar also offers the specification and solution of IDE problems.

8. Recapitulation

To round up this thesis, in the last chapter, after the observations of chapter 5 and chapter 6, it is worth collecting further ideas to better suit non-trivial programs at a wider range of property tests while highlighting already existing limitations to possibly combine speed and completeness in future work.

8.1. Future work

As the choice between the two IFDS problem models is currently driven by the mutually exclusive demand for features and an acceptable performance for non-trivial programs, there is a need to investigate the modeling of property tests and working on existing limitations more deeply. As shown by section 5.1, any choice to implement a static analysis using Phasar is currently accompanied by polynomial costs.

In this section, limitations to any of the presented IFDS problem models as well as their actual implementations are being discussed in detail with possible solutions being proposed. Some topics can be considered a threat to completeness of verification or falsification of MPI RMA programs, some as attempts to improve the run time, and some as justification of skipping certain MPI RMA property tests so far, and thus serve as potential subjects to future work.

LLVM IR

With more than 400 items, the intermediate representation of LLVM provides a rich set of instructions, types and intrinsic functions, richer than other common IRs such as *Vex IR* by Valgrind [Mä17]. While LLVM IR puts great effort into balancing the availability of platform-specific exploits of features and reusability of the IR, as a consequence, any tool claiming completeness over the LLVM IR must adopt the IR specification completely.

As some considerations have been addressed by the implementation in subsection 4.1.2 already, technically, the IR still provides features of general relevance to the IFDS problem handling that have been omitted by the current implementation.

For example, the implementation is able to trace uses of data inside of *aggregate* types of the IR (arrays, structs) but not when loaded and used as an SIMD vector element. As a consequence, when using highly optimized code, the static analysis fails to recognize local or buffer actions on such data.

In another observation, when using the graph slicing algorithm of subsection 5.2.2, LLVM may complain about the incorrect – or worse – missing specification of symbols to consider in a *ϕ-instruction*: For an SSA-style language, a ϕ-instruction indicates a path-sensitive choice between symbols that are mutually exclusive over an actual predecessor *basic block* (BB) that

reached that statement. For the plain purpose of a static analysis, an incomplete set of symbols in a ϕ-instruction is not disrupting to the CFG, yet it may have an impact to the soundness in some cases, rendering the graph slicing inappropriate.

To fully achieve completeness from the perspective of the IR, every single IR instruction relevant for the control flow and data flow must be addressed by the various steps in the implementation. While the IR by LLVM has a respectable size, it is still an acceptable tradeoff compared to handling various programming languages individually.

Dynamic window memory deallocation

In the context of MPI RMA, for windows that are subject to the dynamic flavor, MPI provides the methods `MPI_Win_attach` and `MPI_Win_detach` to adjust the amount of available memory at the window's lifetime. While the implementation is able to trace the base address of memory blocks that have been attached to a window by a call to `MPI_Win_attach` for the purpose of identifying local actions via `Win` facts, there is currently no modeling of retracting memory locations for instructions following after a call to `MPI_Win_detach`.

```
1  MPI_Win_attach(win, data, ...);
2  MPI_Win_attach(win, data2, ...);
3
4  uint8_t *slice = data2;
5
6  MPI_Win_detach(win, data2);
7
8  while (...) {
9    uint8_t zero = slice[0];
10
11   slice += step;
12 }
```

```
1  %3 = alloca i8*, align 8
2  %4 = alloca i8*, align 8
3  %6 = alloca i8*, align 8
4  %8 = load i8*, i8** %3, align 8
5  %9 = call i32 @MPI_Win_attach(..., i8* %8, ...)
6  %11 = load i8*, i8** %4, align 8
7  %12 = call i32 @MPI_Win_attach(..., i8* %11, ...)
8  %13 = load i8*, i8** %4, align 8
9  store i8* %13, i8** %6, align 8
10 ; ...
11 %20 = load i8*, i8** %4, align 8
12 %21 = call i32 @MPI_Win_detach(..., i8* %20)
13 %23 = load i8*, i8** %6, align 8
14 %24 = getelementptr inbounds i8, i8* %23, i64 0
15 %25 = load i8, i8* %24, align 1
16 ; ...
```

Listing 8.1: Aliasing of dynamic window memory (left), IR representation (right).

In the example given by listing 8.1, memory is attached to a window twice, and once detached. The pointer variable `slice` (`%6`) is an alias of `data2` (`%4`), used to iterate over sections of the window memory. After removing `data2` from the window memory and performing a synchronization (not shown by the listing), `%6` is still part of the known, local memory base addresses: The first dereferencing of `slice` (`%23` to `%25`) is then considered a local action.

As a consequence of this gap, a fact may wrongly accept a local memory access as a local action that is actually no longer part of the window memory, possibly resulting in false positives in the inter-rank analysis of concurrent local actions.

To overcome this limitation, a `Win` fact has to group the set of local base address aliases by the one that has been used by the call to `MPI_Win_attach` to allow the removal of the set of invalidated symbols on the occurrence of a call to `MPI_Win_detach` by any such symbol. The implementation was postponed as it requires an additional layer of storage and look-up across the path sensitivity of `Win` facts, with its run time maintenance costs probably outweighing

false positives for a rather dynamic use case, only supported by one window flavor.

MPI handle reassignment

Currently, any `Win` or `Comm` fact assumes that memory locations holding any of its handles (`MPI_Comm`, `MPI_Win`) never get lost to a nullification or an override by another fact: Practically, this may result in a leak of MPI resources, possibly making a process hang at a call to `MPI_Finalize` [For15, section 8.7]. In contrast, for the static analysis, a respective fact, that has been robbed its handle before reaching a call to a proper MPI cleanup method, continues to erroneously assume ownership on the occurrence of future history items.

This circumstance becomes an issue even by non-erroneous use of handle variables if the data flow of the program is modeled by reassigning handles to global or module variables instead of passing them by value to a procedure. Facing an increased risk of races or stalls by this programming pattern in the context of a multi-threaded environment, there may be valid reasons for such a design.

```
1  MPI_Win win1, win2;
2
3  MPI_Win_create(..., &win1);
4  MPI_Win_create(..., &win2);
5
6  win2 = win1;
7
8  MPI_Win_free(&win1);
9  MPI_Win_free(&win2);
```

Listing 8.2: Losing an `MPI_Win` handle.

In the source code given by listing 8.2, the implementation will by-catch the error secondarily: Given two `Win` facts representing `win1` and `win2`, `win2` becomes a handle-copy for `win1` eventually. As a consequence, the original fact of `win2` will continue to record its history until `MPI_Win_free` whereas `win1` will encounter two calls to `MPI_Win_free`, resulting in a violation of the grammar specified by subsection 4.2.1.

To solve this problem, all handle-maintaining fact types have to scan for `store` instructions to known memory locations, invalidating these locations and all of their `load`s inside of the current procedure frame.

Request-based MPI communication

For MPI windows in an access epoch of the *passive target synchronization* (PTS), request-based remote actions are allowed: `MPI_Raccumulate`, `MPI_Rget`, `MPI_Rget_accumulate` and `MPI_Rput`. For each of these functions, the user is required to provide an `MPI_Request` handle that is used to later query and await the state of the local consistency of the originating remote action. The request-terminating functions include `MPI_Wait{all/any/some}` and `MPI_Test{all/any/some}`.

While calls to `MPI_Wait` or `MPI_Test` are fairly simple to implement in the context of an IFDS problem, whether as part of a `Win` fact or as a distinct `Request` fact type, the derivatives

of the suffices `all`, `any` and `some` are non-trivial and require a well-defined semantics: These methods accept a list of `MPI_Request` objects and return the completion state if all, one, or any non-zero number of requests have completed. Aside from properly monitoring the list composition and use of the respective request handles, this imposes the following challenge: On the occurrence of a call to `MPI_{Test/Wait}{any/some}`, how can a static analysis generally assume the effect or voidness of local consistency after such a query? A request may or may not be locally synchronized after a return, thus any local or buffer action directly following may or may not be locally consistent to the remote action. Consequently, both outcomes have to be modeled.

An additional question arises from the use of `MPI_Test`: When moving this call into a loop, the host process can be designed to continue working on other operations in the meantime until reaching completion. While a single call to `MPI_Test` may or may not return completion of a request, repeatedly calling this method and correlating the return to the loop's exit condition, then after leaving the loop, any subsequent buffer action can be considered synchronized. An example of this case is given by listing 8.3.

```
1    int doneWaiting = 0;
2    MPI_Request request;
3    MPI_Rget(..., &request);
4
5    MPI_Test(request, &doneWaiting, ...);
6    while (0 == doneWaiting) {
7      // ... (do other operations)
8      MPI_Test(request, &doneWaiting, ...);
9    }
```

Listing 8.3: Active polling using `MPI_Test`.

The IFDS problem model requires modifications to reliably identify correlations of the return value of a call to `MPI_Test` to the evaluation of a loop condition, allowing the respective fact to acknowledge local consistency of the remote action.

Any modeling of actions related to request-based MPI RMA communication has been skipped for this thesis as partial modeling of this feature yields the unsatisfying consequences and open questions described by the previous paragraphs. Analogously, the RMA-specific method `MPI_Win_test` of the *general active target synchronization* (GATS) synchronization mode is currently ignored.

From a distance of RMA communication, modeling of request handles is also beneficial to the deadlock analysis described in section 4.2.3: As calls to `MPI_Barrier` block the invoking process, a call to `MPI_Ibarrier` defers blocking until the appropriate call to `MPI_Wait`, asking for a more granular modeling in this case.

Context sensitivity

Previously justified by subsection 3.6.1, the IFDS problem solution of Phasar prohibits certain assumptions on other facts while processing a given fact in a flow function: Most notably, any post-generation correlation of two facts must not take place during the execution of the IFDS problem solution. Without revamping the IFDS problem fact domain presented by

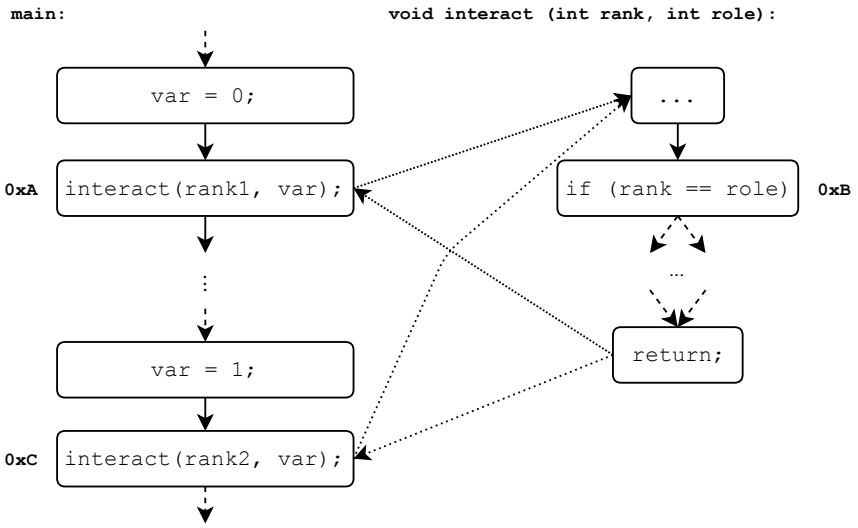


Figure 8.1.: Fact correlation over multiple call sites.

subsection 4.1.3, a minimum degree of building correlations between facts is still desirable, e. g. for the rank attribution.

Yet, the current implementation of fact correlation is limited, losing the state under a given call site context: Consider the example presented by figure 8.1.

In figure 8.1, CFGs of two procedures are shown: `main` and `interact`. For the variable `var`, there exists an `IntConst` fact c_{var}, as well as two facts of type `Null` or `Comm` trace the variables `rank1`, `rank2` for containing the result of a call to `MPI_Comm_rank`. In the CFG of `interact`, there is a comparison of the rank value and `var`, locally known as `role`. c_{var} persists its currently known integer constant into a global map, indexed by `llvm::Instruction*`. If the rank comparison is symbolically located at some location `0xB` throughout the IFDS problem, the map neglects the context sensitivity of the call site, as c_{var} will later overwrite `0xB` $\mapsto 0$ of the first call site (`0xA`) by `0xB` $\mapsto 1$ of the second call site (`0xC`) after `var` has been assigned a new value.

For MPI windows generated by the facts holding `rank1` or `rank2`, it is only correct to correlate the rank and the integer value if both facts can be subsumed to the same context. Similarly, the deadlock test of section 4.2.3 could then be extended to calls to `MPI_Barrier` to arbitrary MPI communicators at any suitable place.

Since context sensitivity requires capturing the necessary closures for an arbitrary number and depth of call stacks, this feature will greatly increase the memory consumption and decrease the performance of the IFDS problem solution and inter-rank tests.

Procedure coloring

In subsection 5.2.2, two DFS-based algorithms were presented that make use of binary node colors of a given CFG, based on simple assumptions of their relevance of contributing to MPI RMA static analysis results. Currently, one of the criteria for a relevant, black node is

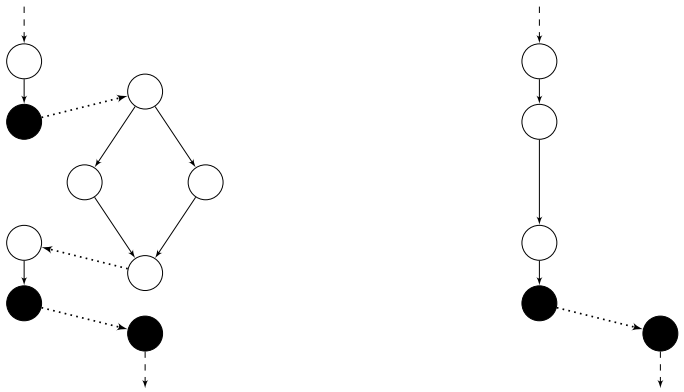


Figure 8.2.: Coloring by the implementation (left), with call site elimination (right).

the presence of a call to a procedure (`llvm::Function`) that is defined by the same IR module.

This may introduce an overly high number of black nodes under the following condition: If the called procedure itself is free of black nodes, then there is no need to mark the caller node black. For every procedure, on the first occurrence of such a call, the setup may iterate the call tree and respective CFGs recursively, caching the overall color state of every procedure for any future reference.

This mechanism will not only reduce the number of black nodes but also allow to completely remove seemingly irrelevant call sites from every CFG if the type of the returned value allows a trivial replacement by a surrogate value. A visual demonstration of the call site elimination by a global CFG color is given by figure 8.2.

Scheduling of intra-path tests

One of the ideas to improve the performance of the implementation while not considering fundamental changes to the IFDS problem model is the embedding of all suitable intra-path tests (subsection 4.2.2) into the online IFDS problem solution (subsection 4.2.1).

Despite considerations of violating best-practices of model and software engineering over separating these concerns, there is no technical inhibition to merge these tests: As a window's flavor is identified by its generating MPI method call, the very first one in every case, tests for illegal attempts of dynamic memory management or queries for shared memory ranges can be embedded into the IFDS problem solution for free, without asking for changes to the grammar of subsection 4.2.1.

With the exception of rank-specific calls to `MPI_Win_flush` or `MPI_Win_unlock`, this is also feasible to a subset of RMA synchronization modes when testing for non-synchronized buffer actions.

Overlapping analysis

When two or more non-synchronized actions target an overlapping memory region, and at least one performs a write, then a data race is imminent. While in theory this is a particularly interesting property to test for MPI RMA at the inter-rank analysis stage, it also suffers from a high dimensionality of variables:

w_{ds} : the size of the window's displacement unit, in bytes
c_d : the number of target displacements of a remote action
c_e : the number of target elements of a remote action
c_t : the `MPI_Datatype` of all target elements of a remote action

The tuple (s, e) describing the positions of the first byte s and last byte e of a remote action c inside of an MPI window w is then calculated as follows:

$$(s, e) = (w_{ds} \cdot c_d, w_{ds} \cdot c_d + c_e \cdot \texttt{MPI\_Type\_size}(c_t))$$

Even if all variables were available as constants, a static analysis would have to correctly follow the setup of a possibly non-basic data type c_t, and to correctly perform the calculation of its size as `MPI_Type_size` would, too. The deriving of types from basic types is covered by chapter 4 of [For15] completely, rendering the whole idea of a static analysis that detects potential risks of overlapping memory operations – balanced to a reasonable degree of warnings or false positives – infeasible for this thesis.

8.2. Conclusion

In this thesis, a tool pipeline to perform a static analysis in the domain of MPI one-sided communication has been evaluated to a satisfying degree: For programming languages commonly found in the context of HPC, LLVM has been proven to act as a convenient intermediate platform to operate a static analysis on, free of drawbacks. A typestate analysis, such as testing an API call sequence for its correctness, can be modeled as an IFDS problem to a certain extent, and given its expressive and mostly one-dimensional design, the specification of MPI RMA fits sufficiently well over its essential parts.

Nevertheless, IFDS problem modeling requires an attentive development if the implementation of facts and flow functions is supposed to lift heavy states and non-trivial use cases. Unless precautions were taken, program structures as simple as a loop may render a problem model useless for its non-finiteness.

Despite the title of this thesis, the implementation is still far away from reaching the rank of a verification tool at the current state, notably for gaps being discussed in section 8.1. Yet, it shows promising approaches to already act as a tool for falsification of software using MPI RMA, following a grammatical specification and heuristics in the post-IFDS problem stage.

The two IFDS problem models presented by chapter 4 and chapter 6 demonstrate how to differently approach the implementation of a verification idea: While the first one aims to

enable a wide range of property tests over the cost of a very limited scalability, the second one attempts to be light and lazy, sacrificing features for a content degree of performance in the online IFDS problem solution of a typestate analysis. Out of the act of necessity to tailor an implementation of the first model that leads to an acceptable run time, steps for optimizations surfaced that benefit both models when facing larger or more complex programs.

For the future development, core topics include the questions of convergence of these two approaches, the modeling of the open topics, the incorporation of techniques to better eliminate infeasible execution paths, or even an evaluation against solutions that use different models, for example an IDE problem model.

8.3. Acknowledgments

A special acknowledgment shall be devoted to the authors of Phasar for the software and their support: For its young age, the framework provides a stable and surprisingly simple environment to quickly model, evaluate and customize an IFDS problem and its implementation – with some background in the LLVM IR programming interface being helpful. With Phasar, the overall acceleration of the progress of this thesis has been substantial.

Bibliography

[AI94] L.O. Andersen and Københavns Universitet. Datalogisk Institut. *Program Analysis and Specialization for the C Programming Language*. DIKU rapport. Datalogisk Institut, 1994.

[AWH16] Mario Arkenberg, Viktoria Wichert, and Peter H. Hauschildt. Proceeding On : Parallelisation Of Critical Code Passages In PHOENIX/3D. In *19th Cambridge Workshop on Cool Stars, Stellar Systems, and the Sun (CS19)*, Cambridge Workshop on Cool Stars, Stellar Systems, and the Sun, page 31, October 2016.

[BA07] Kevin Bierhoff and Jonathan Aldrich. Modular Typestate Checking of Aliased Objects. *SIGPLAN Not.*, 42(10):301–320, October 2007.

[BA08] Kevin Bierhoff and Jonathan Aldrich. Plural: Checking protocol compliance under aliasing. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 971–972, New York, NY, USA, 2008. ACM.

[BBL10] I. S. Bajwa, B. Bordbar, and M. G. Lee. OCL Constraints Generation from Natural Language Specification. In *2010 14th IEEE International Enterprise Distributed Object Computing Conference*, pages 204–213, Oct 2010.

[BCD+18] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.*, 51(3), 2018.

[BH98] E. Baron and Peter H. Hauschildt. Parallel Implementation of the PHOENIX Generalized Stellar Atmosphere Program. II. Wavelength Parallelization. *The Astrophysical Journal*, 495(1):370, 1998.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, Oct 2007.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. *SIGPLAN Not.*, 37(1):1–3, January 2002.

[CFS09] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: A Powerful Approach to Weakest Preconditions. *SIGPLAN Not.*, 44(6):363–374, June 2009.

[DBB+16] James Dinan, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. An Implementation and Evaluation of the MPI 3.0 One-sided Communication Interface. *Concurr. Comput. : Pract. Exper.*, 28(17):4385–4404, December 2016.

[DKL15] Alexander Droste, Michael Kuhn, and Thomas Ludwig. MPI-checker: Static Analysis for MPI. In *Proceedings of the Second Workshop on the LLVM Compiler*

Infrastructure in HPC, LLVM '15, pages 3:1–3:10, New York, NY, USA, 2015. ACM.

[DLS02] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive Program Verification in Polynomial Time. *SIGPLAN Not.*, 37(5):57–68, May 2002.

[For15] Message Passing Interface Forum. *MPI: a Message-passing Interface Standard: Version 3.1.* High-Performance Computing Center, 2015.

[FOSM17] Frank Flederer, Ludwig Ostermayer, Dietmar Seipel, and Sergio Montenegro. Source Code Verification for Embedded Systems using Prolog. In *Proceedings 29th and 30th Workshops on (Constraint) Logic Programming and 24th International Workshop on Functional and (Constraint) Logic Programming, and 24th International Workshop on Functional and (Constraint) Logic Programming, WLP 2015 / WLP 2016 / WFLP 2016, Dresden and Leipzig, Germany, 22nd September 2015 and 12-14th September 2016.*, pages 88–103, 2017.

[HDT+15] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. Remote Memory Access Programming in MPI-3. *ACM Trans. Parallel Comput.*, 2(2):9:1–9:26, June 2015.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, October 1969.

[JKK10] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *J. Comput. Secur.*, 18(5):861–907, September 2010.

[JM09] Ranjit Jhala and Rupak Majumdar. Software Model Checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, October 2009.

[Kin76] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, July 1976.

[KLM+15] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. LTSmin: High-Performance Language-Independent Model Checking. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 692–707, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[Mä17] Florian Märkl. Case Study on LLVM as suitable intermediate language for binary analysis. Technical report, Technische Universität München, 2017.

[NAC99] G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pages 399–410, May 1999.

[NCC99] Gleb Naumovich, Lori A. Clarke, and Jamieson M. Cobleigh. Using Partial Order Techniques to Improve Performance of Data Flow Analysis Based Verification. *SIGSOFT Softw. Eng. Notes*, 24(5):57–65, September 1999.

[NL08] Nomair A. Naeem and Ondrej Lhotak. Typestate-like analysis of multiple interacting objects. *SIGPLAN Not.*, 43(10):347–366, October 2008.

[NLR10] Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical Extensions to the IFDS Algorithm. In Rajiv Gupta, editor, *Compiler Construction*, pages 124–144, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[SF09] Manu Sridharan and Stephen J. Fink. The complexity of andersen's analysis in practice. In Jens Palsberg and Zhendong Su, editors, *Static Analysis*, pages 205–221, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[SHB19] Philipp D. Schubert, Ben Hermann, and Eric Bodden. PhASAR: An Inter-Procedural Static Analysis Framework for C/C++. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2019. To appear.

[Squ16] Jannek Squar. MPI-3 algorithms for 3D radiative transfer on Intel Xeon Phi coprocessors. Master's thesis, Universität Hamburg, 10 2016.

[s.r18] JetBrains s.r.o. The State of Developer Ecosystem Survey in 2018. `https://www.jetbrains.com/research/devecosystem-2018/cpp/`, 2018. [last seen: 2018-12-04].

[SRH96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.*, 167(1-2):131–170, October 1996.

[Str83] Robert E. Strom. Mechanisms for Compile-time Enforcement of Security. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 276–284, New York, NY, USA, 1983. ACM.

[Tar71] R. Tarjan. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 114–121, Oct 1971.

[WH06] H. Jin Q. Gao D. Panda W. Huang, G. Santhanaraman. Design and Implementation of High Performance MVAPICH2: MPI2 over InfiniBand . In *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, pages 43–48, May 2006.

[You66] D. H. Younger. Context-free language processing in time n3. In *7th Annual Symposium on Switching and Automata Theory (swat 1966)*, pages 7–20, Oct 1966.

[Zak11] Alon Zakai. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 301–312, New York, NY, USA, 2011. ACM.

A. Usage instructions

A.1. Requirements

The static analysis tool has been tested extensively under Linux. Although platform dependency is supposed to be optional, Linux is currently recommended. The following software is required to be present on the computer when building the tool from the source code:

- *CMake* v3.10 or newer,
- a C++-14-compliant compiler,
- *git*,
- optionally, the MPI development package of the software to inspect.

A.2. Setup

The following steps are mandatory to obtain a working binary program of the tool:

1. Obtain the source code of Phasar[1], check out git tag `v1218` and install all of the required dependencies.
2. Build and install Phasar in its *Release* configuration.
3. Copy the tool's source code from the given media into a local directory and move there by a system shell.
4. Execute: `$ mkdir build && cd build && cmake ..`
5. Check the output for matching dependencies and desired versions. When satisfied, run `$ make` (or the tool that CMake has set up).

A.3. Tests

In order to run the test suite, a *Ruby* installation of v2.5 or newer is required. The *RSpec* package is necessary and can be obtained by running `$ gem install rspec`. Once done, tests can be started by launching a CMake target (usually `$ make test`), or directly by entering the `tests/` subdirectory and running `$ rspec test_suite.rb`.

[1] `https://github.com/secure-software-engineering/phasar`

A.4. Usage

In the most general case, any LLVM IR file can be used directly with the created binary. Refer to the next section for options provided by additional tool flags.

Running the tool, a decent level of activity is being printed to the output by default. After finishing, there are up to three different CSV files in the current directory:

- `windows.csv`: This file contains a list of window paths in the following columns:
  - `win_id`: This column groups instructions by the actual path identifier, with individual items sorted in ascending order by occurrence on the path.
  - `win_state`: The last known state of the window at the time of termination. It may be `valid`, or `violation` for having identified a violation no later than the last instruction of the current ID, or `unfinished` if no proper shutdown call was identified on this path.
  - `rank_pred`: If the path could be narrowed to match a certain rank criteria, this column contains a C-style operator as the predicate. `n/a` on failure or if the rank analysis is disabled.
  - `rank_value`: If the path could be narrowed to match a certain rank criteria, this column contains the integer value of the rank. Example: If `rank_pred: >=, rank_value: 1`, this path belongs to ranks greater or equal 1. To be ignored if the value of `rank_pred` is `n/a`.
  - `code_location`: If debug information is provided by the given IR file, the column shows the source code file name and a line number where to find the original code line.
  - `llvm_location`: This column prints a human-friendly LLVM IR instruction that has been identified as a path-relevant MPI action.

- `violations.csv`: This file lists any post-analysis violations or discards, i. e. FSM violations by illegal arrangement of instructions visible in `windows.csv` are *not* listed here, despite their `violation` status. These columns are given:
  - `win_id`: This is the path identifier where the violation has been observed; please refer to `windows.csv` for a complete look-up by this value.
  - `violation`: The violation code by any internal analysis.
  - `code_location`: If debug information is provided by the given IR file, the column shows the source code file name and a line number where to find the original code line.
  - `llvm_location`: This column prints a human-friendly LLVM IR instruction that has been identified as a path-relevant MPI action.

- `statistics.csv`: A machine-readable format containing the same number of statistics as given in the activity output of the preflight. Primarily intended for test purposes.

A.5. Flags

When using the static analysis, several customizations of the analysis may come handy. Currently, the following flags can be passed to the tool:

Flag	Description
`-B1`	Running in benchmark mode *B1*. Skips writing anything to the CSV files except the header row.
`-B2`	Running in benchmark mode *B2*. Includes the effect of `-B1` and skips any post-analysis passes.
`-d`	Debug output. If set, Phasar prints its IFDS problem detail output, suitable for inspecting fact lifetimes over individual instructions.
`-e ENTRY_POINT`	Entry point, defaults to `main`. When entering the module anywhere but `main`, then override this by `ENTRY_POINT`. For Fortran code, setting `-e MAIN_` is likely to be required.
`-h`, `-help`	Prints an essential usage help to the output.
`-L`	Lazy mode IFDS problem model. Switches to the lazy IFDS problem path sensitivity, currently skipping the post-IFDS problem analysis stage.
`-o PREFIX`	Output prefix, empty by default. Set `PREFIX` to some value if any CSV file is supposed to be created in a path prefixed by `PREFIX`.
`-OI`	Optimization, assuming static rank values. If set, integer assignments are considered immutable, resulting in a decrease of facts to generate. Do not use if rank roles are reassigned.
`-OR`	Optimization, skip the inter-rank analysis passes. If set, no actions are performed to attempt assigning paths to rank predicates, and thus no inter-rank steps will take place.
`-OW`	Optimization, wormhole mode. If set, the tool tries to eliminate sections from the CFG that do not contribute to any analysis. This is a lossy optimization, but it may speed up the run time a lot while possibly losing information such as local memory operations. Refrain from using the option on the printing of warnings about invalid IR.
`-s`	Preflight only. If set, stop after the initial analysis pass. Useful for obtaining statistics to decide on further processing.
`-S`	Skip preflight. No statistics are collected.
`-v`	Prints version information, and also the MPI environment information if it was built against some.
`-V`	Verbose progress. Linux only, displays a progress counter for each created and finalized window during the IFDS problem reachability analysis. Cannot be used together with `-S`.
`-w i`	Window isolation. A `Win` fact is generated on the `i`-th occurrence of a window setup call only.

A.6. Violations

In the file `violations.csv`, the column `violation` encodes the violation that has been identified for the given instruction. These codes are given a description below:

Code	Description
`concurrent_local_access`	A local action takes place at the same time when some other rank may trigger a virtual action here. Under the separate memory model, this is not allowed to happen, regardless of guarantees for non-overlapping.
`elimination_by_rank`	This is not a violation but the information that a window path described in `windows.csv` is no longer valid for passing through contradicting rank predicates.
`fence_flags_mismatch`	For corresponding calls to `MPI_Win_fence`, there is a mismatch in the flags with respect to `MPI_MODE_NOPRECEDE` or `MPI_NODE_NOSUCCEED`. This violation can be identified only if the analysis tool was built against the correct MPI implementation.
`illegal_by_other_rank`	A call to `MPI_Win_lock_all` while there is at least one path that is not lockable at that time.
`illegal_dynamic_memory_use`	A call to `MPI_Win_attach`, `MPI_Win_detach` or `MPI_Win_shared_query` that is illegal under the setup call of the window.
`no_corresponding_rank`	A call for entering an access epoch without a necessary window path that serves as a target.
`unsafe_local_memory_access`	This violation indicates a conflict involving a buffer action.
`waits_forever`	A call to `MPI_Barrier` or `MPI_Win_fence` without a corresponding call on every other path of this window or communicator.
`window_flavors_incompatible`	Not all windows at the same level uniformly follow the shared or non-shared flavor.

List of Acronyms

API *application programming interface*

AST *abstract syntax tree*

BB *basic block*

CFG *control flow graph*

DFS *depth-first search*

DSL *domain specific language*

FSM *finite-state machine*

GATS *general active target synchronization*

GCC *GNU Compiler Collection*

HPC *high-performance computing*

IDE problem *inter-procedural, distributive environment problem*

IFDS problem *inter-procedural, finite, distributive, subset problem*

IR *intermediate representation*

LLVM *Low Level Virtual Machine*

MPI *Message Passing Interface*

MPI RMA *MPI Remote Memory Access*

OMB *OSU Micro-Benchmarks*

OSU *The Ohio State University*

PDA *pushdown automaton*

P/T net *place/transition net*

PTS *passive target synchronization*

RISC *reduced instruction set computer*

SCC *strongly connected component*

SIMD *single instruction, multiple data*

SSA *static single assignment*

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum	Unterschrift

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Ort, Datum	Unterschrift