A Modular LLVM-Based Contract Framework for Parallel Programming Models

Yussur Mustafa Oraji

SCIENTIFIC COMPUTING

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

- Working with language-native parallel programming difficult
  - Different language $\rightarrow$ different API
  - Low-level, little abstraction
- In contrast: Parallel Programming Models
  - MPI [7]
  - OpenSHMEM [3]
  - GASPI [6]
  - ...
- (Sometimes) support for multiple languages
- Very error-prone...

How many issues can you spot in this tiny example?

At least 8 issues in this code example!

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char** argv)
{
    int rank, size, buf[8];

    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    MPI_Datatype type;
    MPI_Type_contiguous (2, MPI_INTEGER, &type);

    MPI_Recv (buf, 2, MPI_INT, size - rank, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send (buf, 2, type, size - rank, 123, MPI_COMM_WORLD);
    printf ("Hello, I am rank %d of %d.\n", rank, size);

    return 0;
}
```

Our Approach

- Current example: MPI, but OpenSHMEM and other APIs are used as well
- Main languages: C/C++ and Fortran
- $\Longrightarrow$ Need tool for each language and programming model...

Our Approach

- Current example: MPI, but OpenSHMEM and other APIs are used as well
- Main languages: C/C++ and Fortran
- $\Longrightarrow$ Need tool for each language and programming model. . .
- Idea: “Move the model out of the tool, apply analysis independent of model”

Contract Basics

- Specify API assumptions in code
- Annotate additional information:
  - Preconditions / Premise: “What holds before this function?”
  - Postcondition: “What holds after it?”

[1] Feature delayed, but work continues [4]

Introduction

Contract Basics

- Specify API assumptions in code
- Annotate additional information:
  - Preconditions / Premise: “What holds before this function?”
  - Postcondition: “What holds after it?”
- Early C++20 draft[1]: `expects`, `ensures`, later `pre`, `post`
- Also available in other languages:
  - Prusti Project for Rust [1]
  - Kotlin Contracts [12]

[1] Feature delayed, but work continues [4]

Applying Contracts

- Possible solutions using contracts (pseudocode):

Applying Contracts

- Possible solutions using contracts (pseudocode):
  - `<ret-type> MPI_* PRE{ call!(MPI_Init) }` $\Longrightarrow$ Ensures `MPI_Init` called
  - ```
    int MPI_Type_contiguous(..., MPI_Datatype* type) POST{
    no!  (call!(MPI_Irecv,*type))
    until!  (call!(MPI_Type_commit,type))}
    ```
    $\Longrightarrow$ Ensures type committed

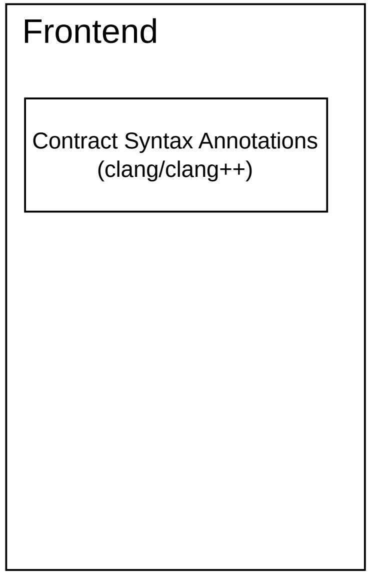
SCIENTIFIC COMPUTING
TECHNISCHE UNIVERSITÄT DARMSTADT
    ```

Applying Contracts

- Possible solutions using contracts (pseudocode):
  - `<ret-type> MPI_* PRE{ call!(MPI_Init) }` ⟹ Ensures `MPI_Init` called
  - `int MPI_Type_contiguous(..., MPI_Datatype* type) POST{`
    `no! (call!(MPI_Irecv,*type))`
    `until! (call!(MPI_Type_commit,type))}` ⟹ Ensures type committed

Analyses unaware of model: No need for additional analyses for other models!

Applying Contracts

- Possible solutions using contracts (pseudocode):
  - `<ret-type> MPI_* PRE{ call!(MPI_Init) }` $\Longrightarrow$ Ensures `MPI_Init` called
  - ```
    int MPI_Type_contiguous(..., MPI_Datatype* type) POST{
    no!  (call!(MPI_Irecv,*type))
    until!  (call!(MPI_Type_commit,type))}
    ```
    $\Longrightarrow$ Ensures type committed

Analyses unaware of model: No need for additional analyses for other models!

Inherently extensible: For other error classes / other models, simply add the annotations!
    ```

Architecture Considerations

Architecture Considerations

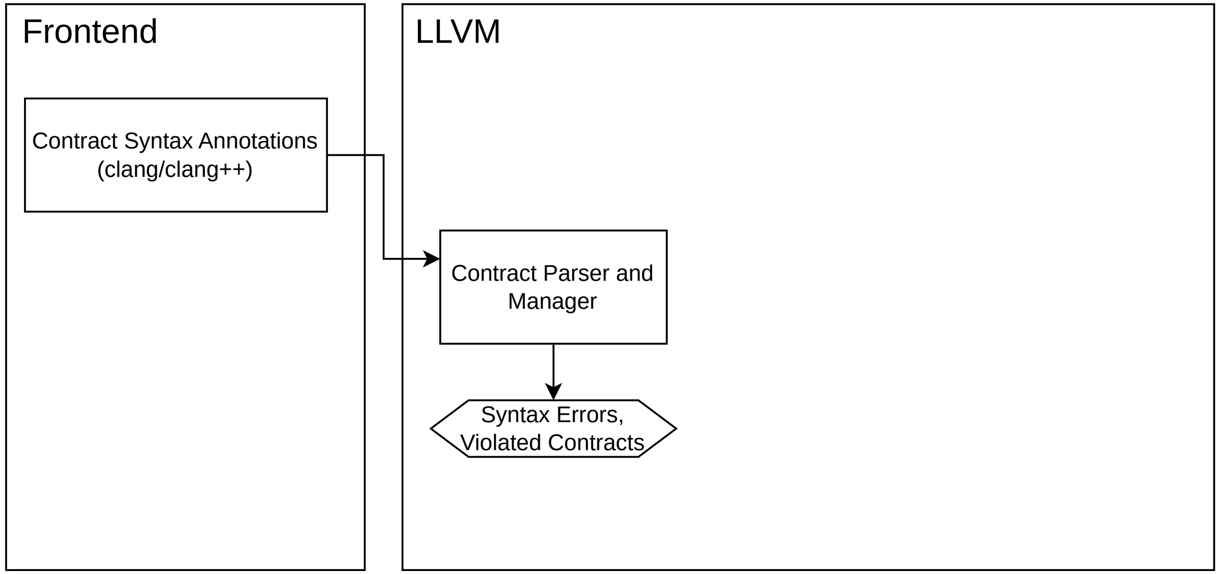
- For Proof of Concept:

Architecture Considerations



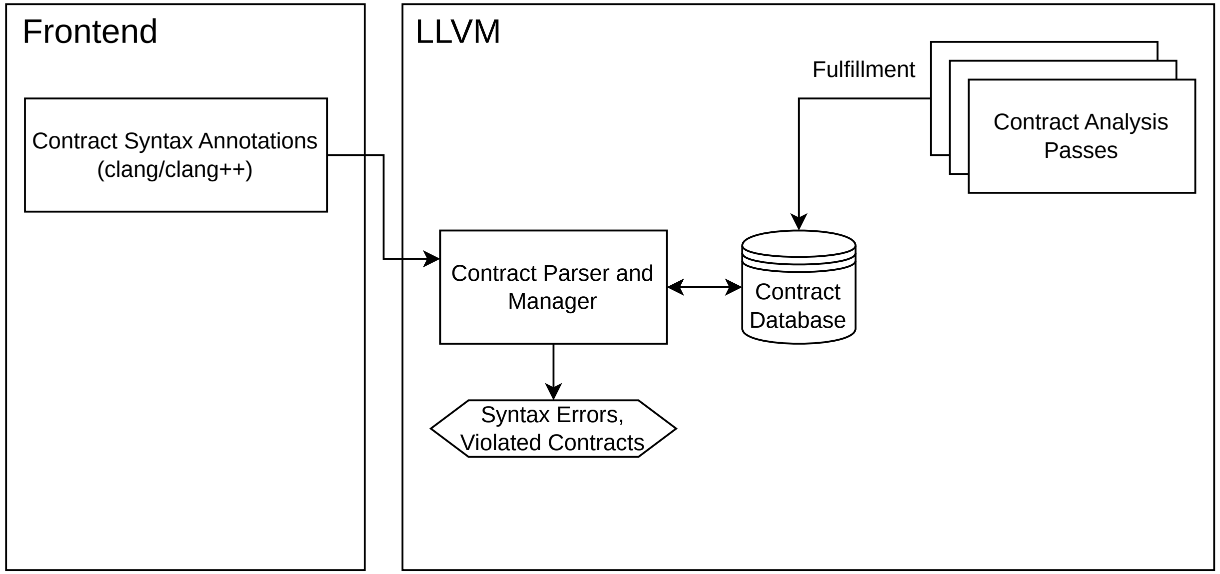
- For Proof of Concept:
  1. Develop contract syntax for considered languages $\Longrightarrow$ Function Annotations

Architecture Considerations



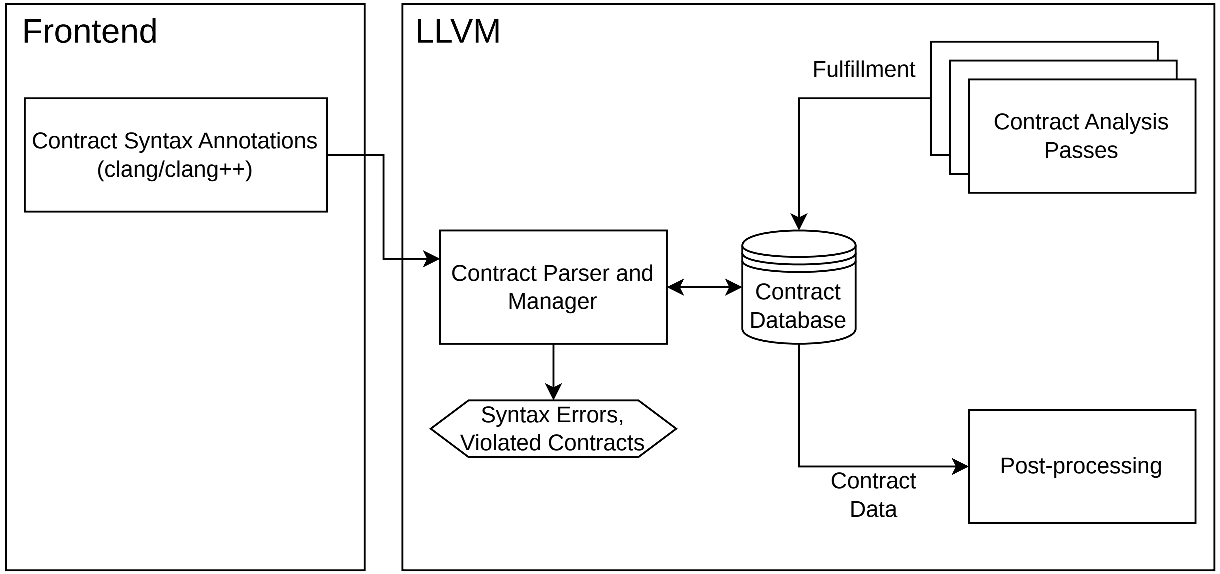
- For Proof of Concept:
  1. Develop contract syntax for considered languages $\implies$ Function Annotations
  2. Contract Parser and Manager as LLVM Analysis

Architecture Considerations



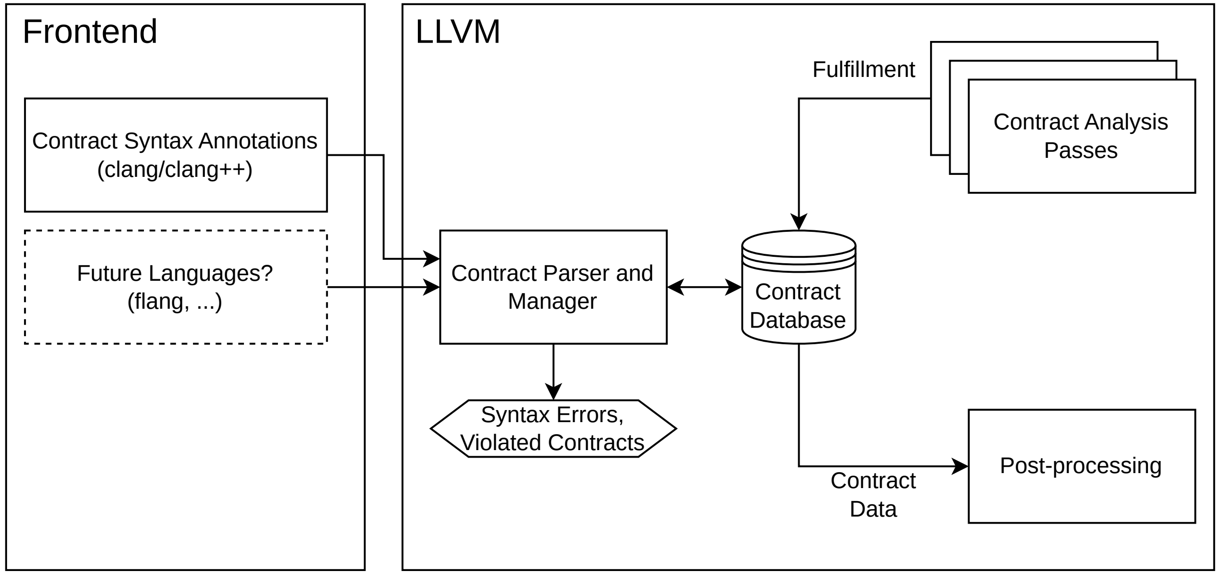
- For Proof of Concept:
  1. Develop contract syntax for considered languages $\implies$ Function Annotations
  2. Contract Parser and Manager as LLVM Analysis
  3. Contract Analysis Pass(es) mark contracts as fulfilled, unable to prove, or violated in manager database

Architecture Considerations



- For Proof of Concept:
  1. Develop contract syntax for considered languages $\Longrightarrow$ Function Annotations
  2. Contract Parser and Manager as LLVM Analysis
  3. Contract Analysis Pass(es) mark contracts as fulfilled, unable to prove, or violated in manager database
  4. Post-processing performs last steps before reporting issues if present

Architecture Considerations



- For Proof of Concept:
  1. Develop contract syntax for considered languages $\Longrightarrow$ Function Annotations
  2. Contract Parser and Manager as LLVM Analysis
  3. Contract Analysis Pass(es) mark contracts as fulfilled, unable to prove, or violated in manager database
  4. Post-processing performs last steps before reporting issues if present

Error Classes and Detection Methods

- Currently focusing on the following errors:

Error Classes and Detection Methods

- Currently focusing on the following errors:
  1. Request Lifecycle errors: Request not checked or request leaked
     - Request reuse before completion leaks memory

```
MPI_Request req;
MPI_Isend(&data, ..., &req);
MPI_Isend(&data, ..., &req);
// Only 2nd call is completed!
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Error Classes and Detection Methods

- Currently focusing on the following errors:
  1. Request Lifecycle errors: Request not checked or request leaked
  2. Resource (Datatype, Communicator) not initialized or freed
     - Many kinds of resources must be managed by the user manually
     - If not freed, data leaks occur. If not initialized, crashes or undefined behavior

```
MPI_Datatype type;
MPI_Type_contiguous(..., &type);
MPI_Send(&data, 1, type, ...);
```

Error Classes and Detection Methods

- Currently focusing on the following errors:
  1. Request Lifecycle errors: Request not checked or request leaked
  2. Resource (Datatype, Communicator) not initialized or freed
  3. Local Data Race
     - Buffers of nonblocking operations may not be written to, and sometimes also not read

```c
int value;
MPI_Win_fence(0, win);
if (rank == 0) {
    MPI_Get(&value, 1, ..., win);
    printf("Value is: %d", value);
}
MPI_Win_fence(0, win);
```


SCIENTIFIC COMPUTING
TECHNISCHE UNIVERSITÄT DARMSTADT

Error Classes and Detection Methods

- Currently focusing on the following errors:
  1. Request Lifecycle errors: Request not checked or request leaked
  2. Resource (Datatype, Communicator) not initialized or freed
  3. Local Data Race
  4. Missing Init/Finalize

Error Classes and Detection Methods

- Currently focusing on the following errors:
  1. Request Lifecycle errors: Request not checked or request leaked
  2. Resource (Datatype, Communicator) not initialized or freed
  3. Local Data Race
  4. Missing Init/Finalize
  5. Unmatched wait, Mixed RMA sync or outside epoch, . . .

Error Classes and Detection Methods

- Currently focusing on the following errors:
  1. Request Lifecycle errors: Request not checked or request leaked
  2. Resource (Datatype, Communicator) not initialized or freed
  3. Local Data Race
  4. Missing Init/Finalize
  5. Unmatched wait, Mixed RMA sync or outside epoch, . . .

- Error classification leads to planned contract syntax
  - Require function called before/after current: `PRE / POST { call!(`*f*`) }`
  - Require a release operator: `POST { no! (`*op*`) until! (call!(`*f*`)) }`

Error Classes and Detection Methods

- Currently focusing on the following errors:
  1. **Request Lifecycle errors: Request not checked or request leaked**: `POST { call!(MPI_Wait) }`
  2. Resource (Datatype, Communicator) not initialized or freed
  3. Local Data Race
  4. **Missing Init/Finalize**: `PRE { call!(MPI_Init) }`
  5. Unmatched wait, Mixed RMA sync or outside epoch, ...

- Error classification leads to planned contract syntax
  – **Require function called before/after current:** `PRE / POST { call!(`*f*`) }`
  – Require a release operator: `POST { no! (`*op*`) until! (call!(`*f*`)) }`

Error Classes and Detection Methods

- Currently focusing on the following errors:
  1. Request Lifecycle errors: Request not checked or request leaked
  2. Resource (Datatype, Communicator) not initialized or freed
  3. **Local Data Race**: `POST { no! (read!(buf)) until! (call!(MPI_Win_fence)) }`
  4. Missing Init/Finalize
  5. Unmatched wait, Mixed RMA sync or outside epoch, ...

- Error classification leads to planned contract syntax
  - Require function called before/after current: `PRE / POST { call!(`*f*`) }`
  - **Require a release operator:** `POST { no! (`*op*`) until! (call!(`*f*`)) }`

SCIENTIFIC COMPUTING

TECHNISCHE UNIVERSITÄT DARMSTADT

Contract Abstraction

- Use ANTLR [14] to define formal grammar
- C/C++: Define contracts as function annotations
  - Helper macro: `#define CONTRACT(x) __attribute__((annotate("CONTRACT{" #x "}")))`

Contract Abstraction

- Use ANTLR [14] to define formal grammar
- C/C++: Define contracts as function annotations
  - Helper macro: `#define CONTRACT(x) __attribute__((annotate("CONTRACT{" #x "}")))`

Annotating using the helper macro:

```
int MPI_Finalize(void)
CONTRACT(PRE {call!(MPI_Init)});
```

Error Classes and Detection Methods

Contract Abstraction

- Use ANTLR [14] to define formal grammar
- C/C++: Define contracts as function annotations
  - Helper macro: `#define CONTRACT(x) __attribute__((annotate("CONTRACT{" #x "}")))`
- Allow defining tags when functions fulfill similar purpose: `MPI_Win_fence` → `TAGS { rma_complete(1) }`

Annotating using the helper macro:

```
int MPI_Finalize(void)
CONTRACT(PRE {call_tag!(initialize)});
```

```
int MPI_Init(...) CONTRACT (TAGS { initialize })
int MPI_Init_thread(...) CONTRACT (TAGS { initialize })
```

Error Classes and Detection Methods

Contract Abstraction

- Use ANTLR [14] to define formal grammar
- C/C++: Define contracts as function annotations
  - Helper macro: `#define CONTRACT(x) __attribute__((annotate("CONTRACT{" #x "}")))`
- Allow defining tags when functions fulfill similar purpose: `MPI_Win_fence` → `TAGS { rma_complete(1) }`

Annotating using the helper macro:

```
int MPI_Put(void* buf, ..., MPI_Win win) CONTRACT(
POST {
    no!
        write!(*0)
        // Do not write to buffer
    until!
        call_tag!(rma_complete,$:7)
        // Until communication was completed for the window
});
```

Contract Abstraction

- Use ANTLR [14] to define formal grammar
- C/C++: Define contracts as function annotations
  - Helper macro: `#define CONTRACT(x) __attribute__((annotate("CONTRACT{" #x "}")))`
- Allow defining tags when functions fulfill similar purpose: `MPI_Win_fence` → `TAGS { rma_complete(1) }`

Annotating using the helper macro:

```
int MPI_Put(void* buf, ..., MPI_Win win) CONTRACT(
POST {
    no!
        write!(*0)
        // Do not write to buffer
    until!
        call_tag!(rma_complete,$:7)
        // Until communication was completed for the window
});
```

Contract Abstraction

- Use ANTLR [14] to define formal grammar
- C/C++: Define contracts as function annotations
  - Helper macro: `#define CONTRACT(x) __attribute__((annotate("CONTRACT{" #x "}")))`
- Allow defining tags when functions fulfill similar purpose: `MPI_Win_fence` → `TAGS { rma_complete(1) }`
- All contracts may be written at *any* function declaration:
  - Modify original model header (e.g. `mpi.h`)
  - Add additional declarations as separate header
  - Add declarations in source for specific analyses only
  - Chef's Choice: Separate header, include using `-include` preprocessor flag

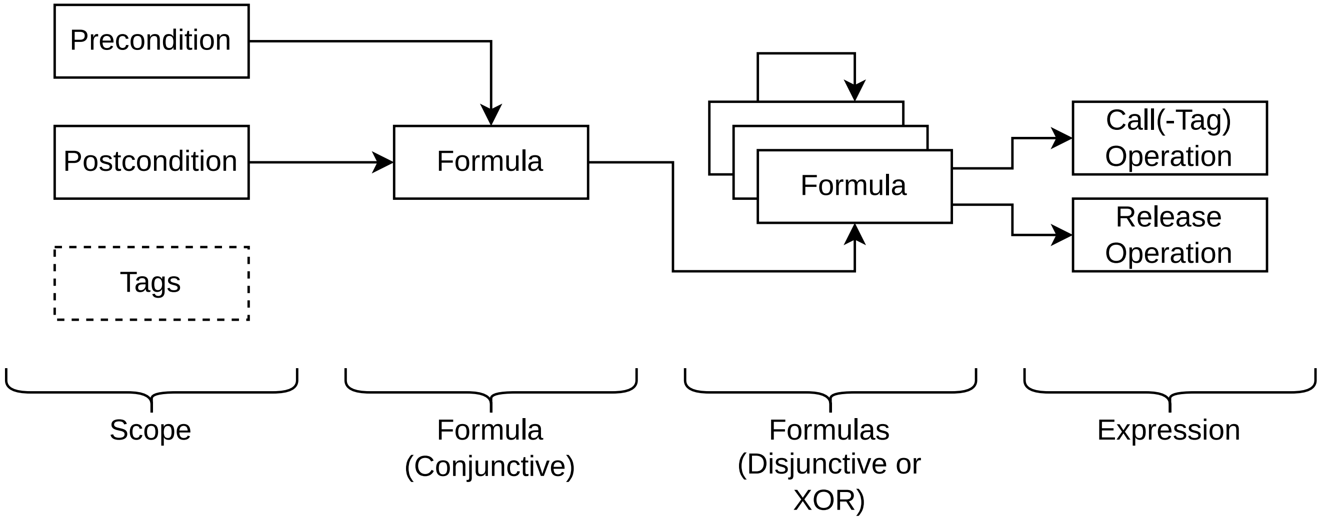
Contract Abstraction

- Contract Manager: LLVM Analysis that reads `@llvm.global.annotations` $\rightarrow$ List of annotations
  - Calls parser, transform contract string into `ContractTree` representation
  - $\implies$ Analyses only interact with `ContractTree`, no contract language details needed

Error Classes and Detection Methods

Contract Abstraction

- Contract Manager: LLVM Analysis that reads `@llvm.global.annotations` $\rightarrow$ List of annotations
  - Calls parser, transform contract string into `ContractTree` representation
  - $\implies$ Analyses only interact with `ContractTree`, no contract language details needed



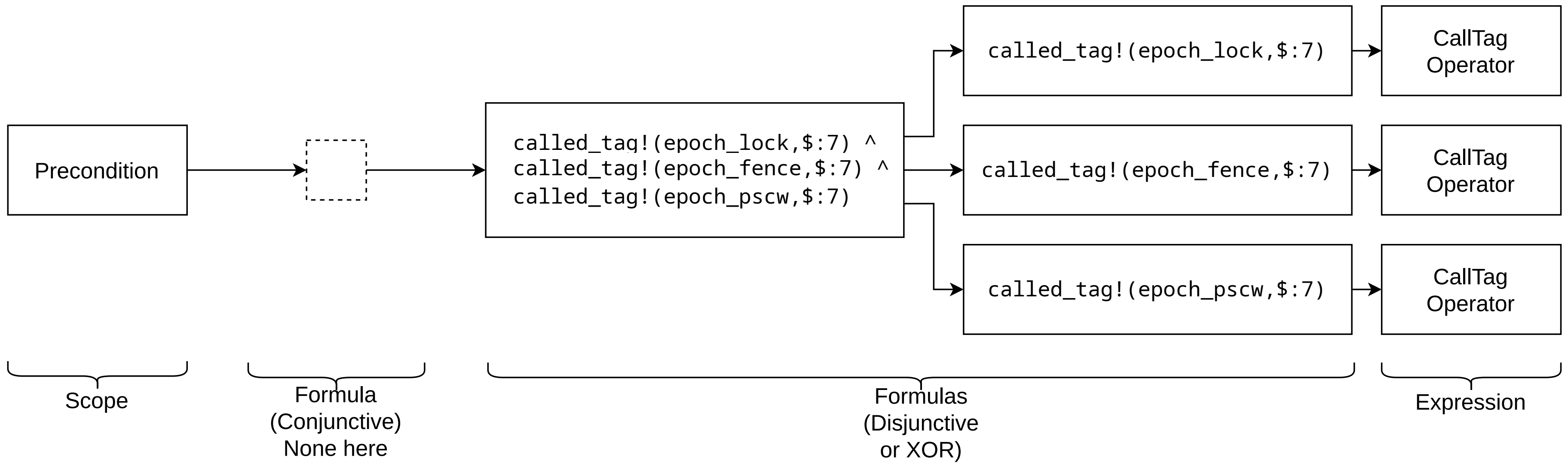
Contract Abstraction

- Contract Manager: LLVM Analysis that reads `@llvm.global.annotations` $\rightarrow$ List of annotations
  - Calls parser, transform contract string into `ContractTree` representation
  - $\implies$ Analyses only interact with `ContractTree`, no contract language details needed
- Example: “For an RMA operation, an epoch should exist (any kind, but not mixed)”

```
int MPI_Put(...) CONTRACT(
PRE {
    call_tag!(epoch_lock,$:7)} ^
    call_tag!(epoch_fence,$:7)^
    call_tag!(epoch_pscw,$:7)
});
```

Contract Abstraction

- Contract Manager: LLVM Analysis that reads `@llvm.global.annotations` $\rightarrow$ List of annotations
  - Calls parser, transform contract string into `ContractTree` representation
  - $\implies$ Analyses only interact with `ContractTree`, no contract language details needed
- Example: "For an RMA operation, an epoch should exist (any kind, but not mixed)"



Error Classes and Detection Methods

Analysis

- Contract Verifiers: Implemented using Data Flow Analysis
  - `PreCall`: Check if function called previously
  - `PostCall`: Check if function called afterward
  - `Release`: Check for conflict before release
- Verifiers share generic worklist algorithm with interprocedural support

Error Classes and Detection Methods

Analysis

- Contract Verifiers: Implemented using Data Flow Analysis
  - `PreCall`: **Check if function called previously**: `MPI_Put` $\rightarrow$ `PRE { call!(MPI_Win_fence) }`
  - `PostCall`: Check if function called afterward
  - `Release`: Check for conflict before release
- Verifiers share generic worklist algorithm with interprocedural support. Current: $\{(1,2),(1,4)\}$

```
1 if (...)
2     MPI_Win_fence(win)
3     x++
4 print(x)
5 MPI_Put(&x, ..., win)
6 x = 1
7 return 0
```

Line	Result
1	**NOTCALLED**
2	CALLED
3	CALLED
4	CALLED
5	CALLED
6	CALLED
7	CALLED

Analysis

- Contract Verifiers: Implemented using Data Flow Analysis
  - `PreCall`: **Check if function called previously**: `MPI_Put` → `PRE { call!(MPI_Win_fence) }`
  - `PostCall`: Check if function called afterward
  - `Release`: Check for conflict before release
- Verifiers share generic worklist algorithm with interprocedural support. Current: $\{(2,3),(1,4)\}$

```
1 if (...)
2     MPI_Win_fence(win)
3     x++
4 print(x)
5 MPI_Put(&x, ..., win)
6 x = 1
7 return 0
```

Line	Result
1	NOTCALLED
2	**PARAMCHECK**
3	CALLED
4	CALLED
5	CALLED
6	CALLED
7	CALLED

Analysis

- Contract Verifiers: Implemented using Data Flow Analysis
  - `PreCall`: **Check if function called previously**: `MPI_Put` $\rightarrow$ `PRE { call!(MPI_Win_fence) }`
  - `PostCall`: Check if function called afterward
  - `Release`: Check for conflict before release
- Verifiers share generic worklist algorithm with interprocedural support. Current: $\{(4,5),(1,4)\}$

```
1 if (...)
2     MPI_Win_fence(win)
3     x++
4 print(x)
5 MPI_Put(&x, ..., win)
6 x = 1
7 return 0
```

Line	Result
1	NOTCALLED
2	PARAMCHECK
3	**PARAMCHECK**
4	**PARAMCHECK**
5	CALLED
6	CALLED
7	CALLED

SCIENTIFIC COMPUTING

TECHNISCHE UNIVERSITÄT DARMSTADT

Error Classes and Detection Methods

Analysis

- Contract Verifiers: Implemented using Data Flow Analysis
  - `PreCall`: **Check if function called previously**: `MPI_Put` → `PRE { call!(MPI_Win_fence) }`
  - `PostCall`: Check if function called afterward
  - `Release`: Check for conflict before release
- Verifiers share generic worklist algorithm with interprocedural support. Current: $\{(4, 5)\}$

```
1 if (...)
2     MPI_Win_fence(win)
3     x++
4 print(x)
5 MPI_Put(&x, ..., win)
6 x = 1
7 return 0
```

Line	Result
1	NOTCALLED
2	PARAMCHECK
3	PARAMCHECK
4	**NOTCALLED**
5	CALLED
6	CALLED
7	CALLED

Error Classes and Detection Methods

Analysis

- Contract Verifiers: Implemented using Data Flow Analysis
  - `PreCall`: **Check if function called previously**: `MPI_Put` $\rightarrow$ `PRE { call!(MPI_Win_fence) }`
  - `PostCall`: Check if function called afterward
  - `Release`: Check for conflict before release
- Verifiers share generic worklist algorithm with interprocedural support. Current: $\emptyset$

```
1 if (...)
2     MPI_Win_fence(win)
3     x++
4 print(x)
5 MPI_Put(&x, ..., win)
6 x = 1
7 return 0
```

Line	Result
1	NOTCALLED
2	PARAMCHECK
3	PARAMCHECK
4	NOTCALLED
5	**ERROR**
6	**ERROR**
7	**ERROR**

Analysis

- Contract Verifiers: Implemented using Data Flow Analysis
  - `PreCall`: **Check if function called previously**: `MPI_Put` → `PRE { call!(MPI_Win_fence) }`
  - `PostCall`: Check if function called afterward
  - `Release`: Check for conflict before release
- Verifiers share generic worklist algorithm with interprocedural support

```
1 if (...)
2     MPI_Win_fence(win)
3     x++
4 print(x)
5 MPI_Put(&x, ..., win)
6 x = 1
7 return 0
```

Line	Result
1	NOTCALLED
2	PARAMCHECK
3	PARAMCHECK
4	NOTCALLED
5	ERROR
6	ERROR
7	ERROR

Analysis

- Contract Verifiers: Implemented using Data Flow Analysis
  - `PreCall`: Check if function called previously
  - `PostCall`: Check if function called afterward
  - `Release`: **Check for conflict before release**: `MPI_Put` $\rightarrow$ `POST { no! (write!(*0)) until! ...}`
- Verifiers share generic worklist algorithm with interprocedural support . Current: $\{(5, 6)\}$

```
1 if (...)
2     MPI_Win_fence(win)
3     x++
4 print(x)
5 MPI_Put(&x, ..., win)
6 x = 1
7 return 0
```

Line	Result
1	FULFILLED
2	FULFILLED
3	FULFILLED
4	FULFILLED
5	**FORBIDDEN**
6	FULFILLED
7	FULFILLED

Error Classes and Detection Methods

Analysis

- Contract Verifiers: Implemented using Data Flow Analysis
  - `PreCall`: Check if function called previously
  - `PostCall`: Check if function called afterward
  - `Release`: **Check for conflict before release**: `MPI_Put` $\rightarrow$ `POST { no! (write!(*0)) until! ...}`
- Verifiers share generic worklist algorithm with interprocedural support . Current: $\emptyset$

```
1 if (...)
2     MPI_Win_fence(win)
3     x++
4 print(x)
5 MPI_Put(&x, ..., win)
6 x = 1
7 return 0
```

Line	Result
1	FULFILLED
2	FULFILLED
3	FULFILLED
4	FULFILLED
5	FORBIDDEN
6	**ERROR**
7	**ERROR**

Analysis

- Contract Verifiers: Implemented using Data Flow Analysis
  - `PreCall`: Check if function called previously
  - `PostCall`: Check if function called afterward
  - `Release`: **Check for conflict before release**: `MPI_Put` → `POST { no! (write!(*0)) until! ...}`
- Verifiers share generic worklist algorithm with interprocedural support

```
1 if (...)
2     MPI_Win_fence(win)
3     x++
4 print(x)
5 MPI_Put(&x, ..., win)
6 x = 1
7 return 0
```

Line	Result
1	FULFILLED
2	FULFILLED
3	FULFILLED
4	FULFILLED
5	FORBIDDEN
6	ERROR
7	ERROR

Error Classes and Detection Methods

Analysis

- Contract Verifiers: Implemented using Data Flow Analysis
  - `PreCall`: Check if function called previously
  - `PostCall`: Check if function called afterward
  - `Release`: Check for conflict before release
- Verifiers share generic worklist algorithm with interprocedural support
- Post-processing pass outputs results for each contract
  - First: Resolve Formulas
  - Then: Print state, debug info of contract if violated or unknown

Error Classes and Detection Methods

Analysis

- Contract Verifiers: Implemented using Data Flow Analysis
  - `PreCall`: Check if function called previously
  - `PostCall`: Check if function called afterward
  - `Release`: Check for conflict before release
- Verifiers share generic worklist algorithm with interprocedural support

```
## Contract violation detected! ##
--> Function: MPI_Put
--> Contract: CONTRACT{POST { no! (write!(*0)) ...}}
--> Postcondition Subformula Status: Violated
    --> Formula String: no!(write!(*0))...
    --> Error Info:
        [ContractVerifierRelease] Found store at main:6 which
          is in conflict with MPI_Put at main:5 before release
```


SCIENTIFIC COMPUTING

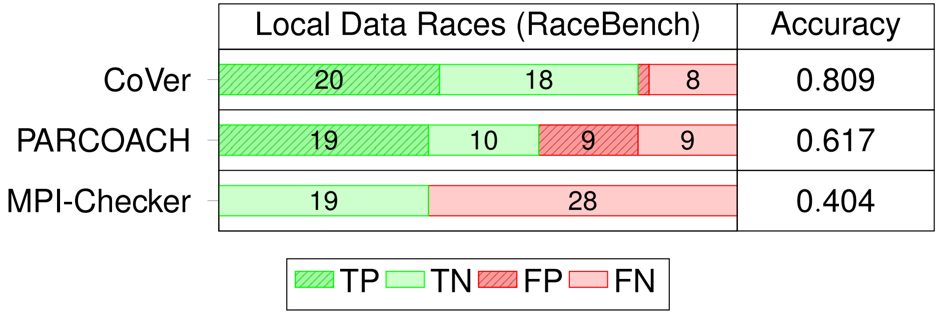
TECHNISCHE UNIVERSITÄT DARMSTADT

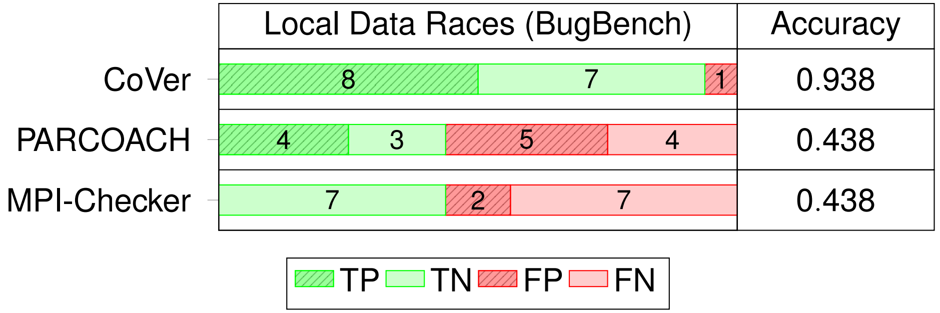
Experiment Setup

- Evaluate both accuracy and compilation overhead
- Accuracy:
  - Tests from MPI-BugBench (Level 1) [10]
  - Tests from RMARaceBench [16]
- Compilation Overhead:
  - LULESH [11]
  - miniVite [8]
  - PRK Stencil [17]
- Compare against PARCOACH-static [15] and MPI-Checker [5]

Evaluation

Accuracy

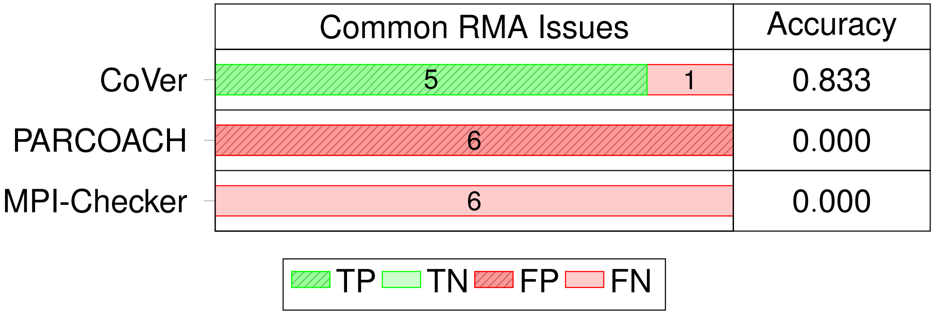


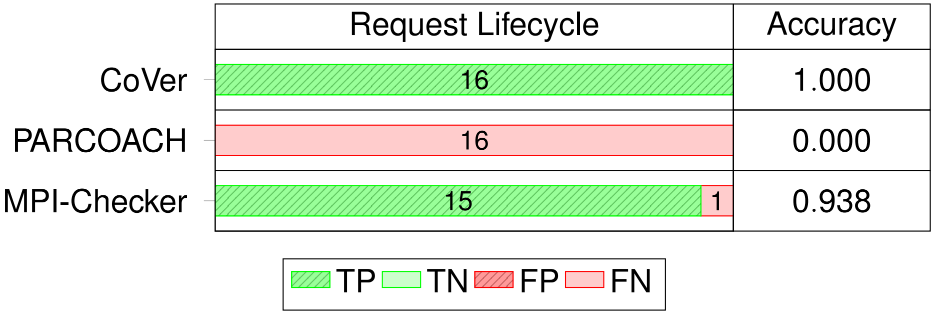


- Focus on only Local Races (direct comparison to PARCOACH)
- Significantly reduced FP rate

Evaluation

Accuracy

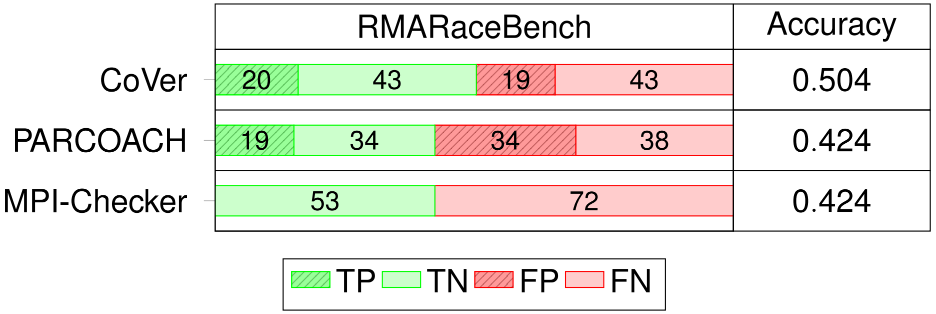


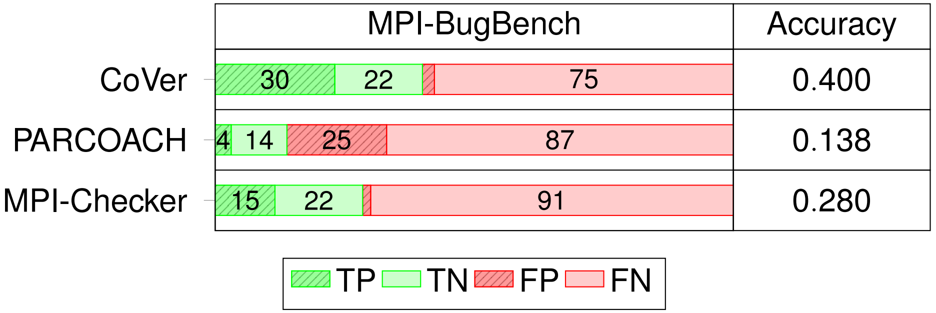


- Focus on only Local Races (direct comparison to PARCOACH)
- Significantly reduced FP rate
- CoVer: Wide error class coverage
  - Adding checks often trivial
- CoVer and MPI-Checker detect similar request lifecycle issues

Evaluation

Accuracy

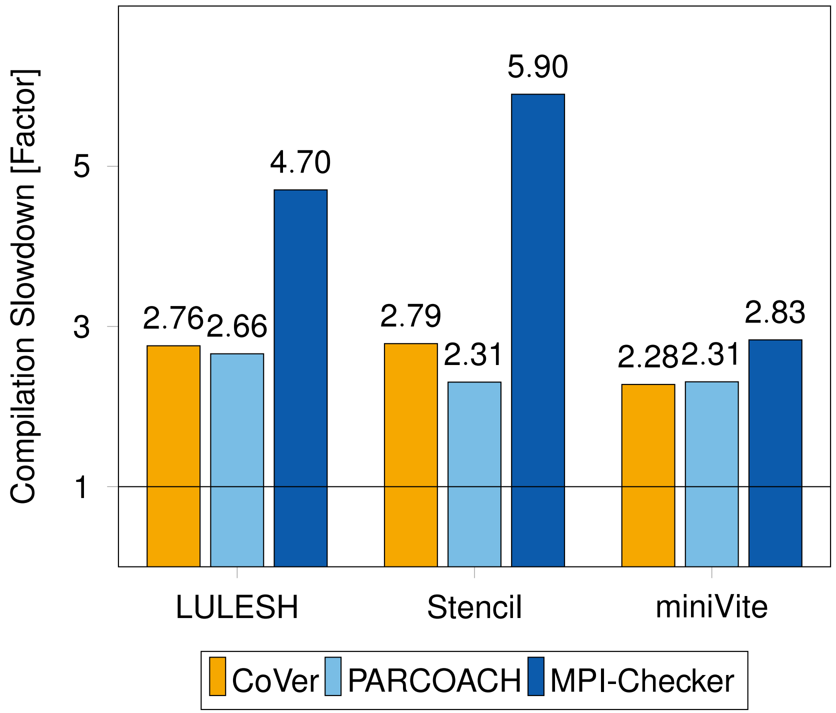




- Focus on only Local Races (direct comparison to PARCOACH)
- Significantly reduced FP rate
- CoVer: Wide error class coverage
  - Adding checks often trivial
- CoVer and MPI-Checker detect similar request lifecycle issues
- Overall: High Accuracy using wide coverage
- But: (In total) Higher FP rate than MPI-Checker
  - Caused by ignoring branch condition

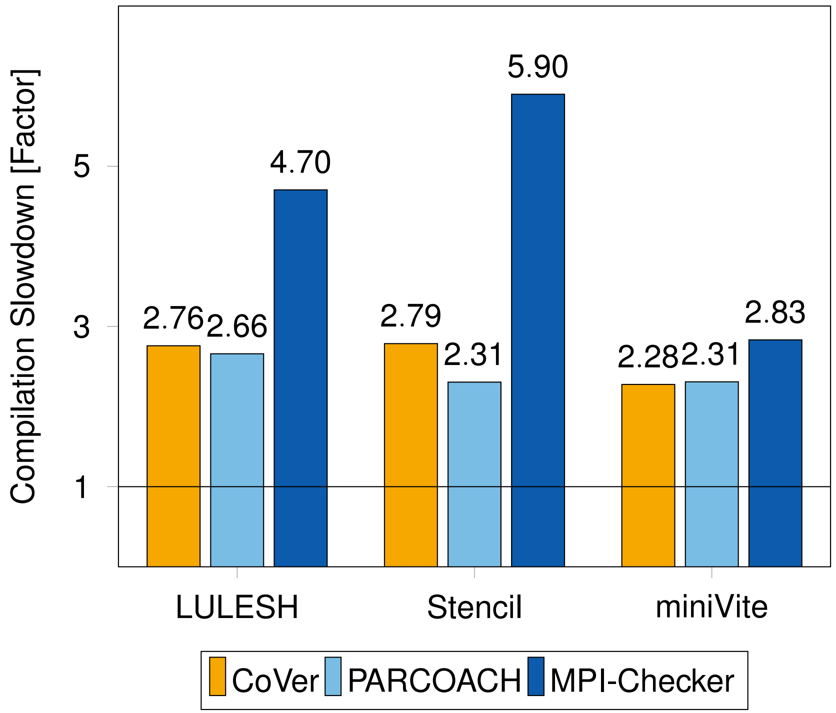
Evaluation

Overhead



- PARCOACH and CoVer have similar runtime

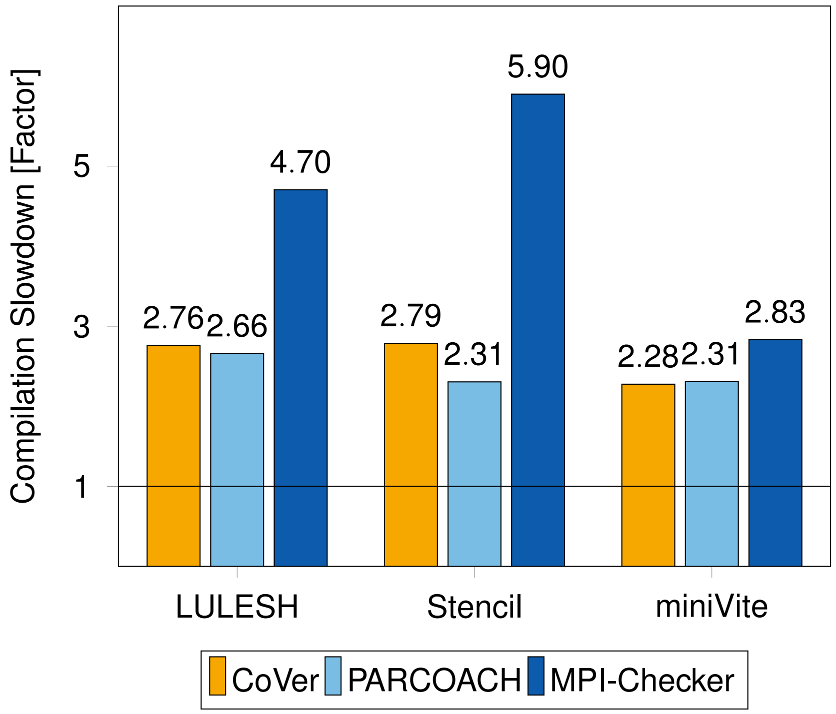
Overhead



- PARCOACH and CoVer have similar runtime
- Note: *All* tools print false positives for these applications
  - Causes: Unknown branch conditions, function pointers, . . .

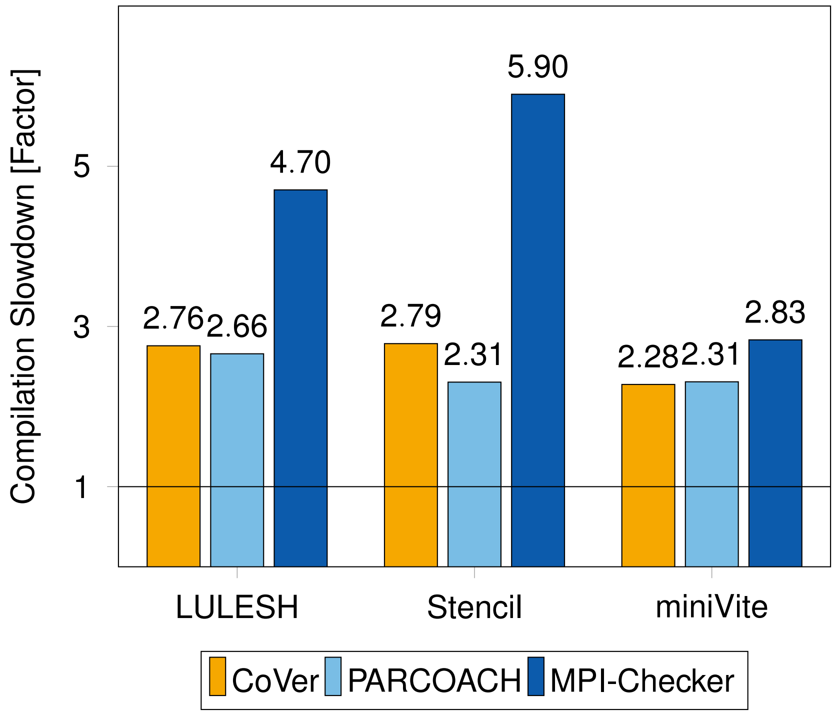
Evaluation

Overhead



- PARCOACH and CoVer have similar runtime
- Note: *All* tools print false positives for these applications
  – Causes: Unknown branch conditions, function pointers, . . .
- MPI-Checker prints the least → Advantage of symbolic execution

Overhead



- PARCOACH and CoVer have similar runtime
- Note: *All* tools print false positives for these applications
  - Causes: Unknown branch conditions, function pointers, ...
- MPI-Checker prints the least → Advantage of symbolic execution
  - Disadvantage shown in the graph...

Related Work

- MUST [9], PARCOACH [15], ITAC [13] are dynamic MPI error detection tools
  - Significant runtime overhead
  - PARCOACH can run statically, but has no support for interprocedural contexts

Related Work

- MUST [9], PARCOACH [15], ITAC [13] are dynamic MPI error detection tools
  - Significant runtime overhead
  - PARCOACH can run statically, but has no support for interprocedural contexts

- MPI-Checker [5] is a static correctness checker
  - Symbolic Execution
  - No support across translation units
  - Bound to `clang` frontend

Related Work

- MUST [9], PARCOACH [15], ITAC [13] are dynamic MPI error detection tools
  - Significant runtime overhead
  - PARCOACH can run statically, but has no support for interprocedural contexts

- MPI-Checker [5] is a static correctness checker
  - Symbolic Execution
  - No support across translation units
  - Bound to `clang` frontend

- SPMD IR [2] is an approach unifying multiple programming models
  - Generic, but non-modular: Cannot extend without source modification
  - Both error classes and supported models static once tool shipped

Conclusion

Tool Summary:

- Ensure modularity by removing model info from analysis
- Use contracts to verify program properties
- Can detect a wide variety of issues, easy to extend

Conclusion

Tool Summary:

- Ensure modularity by removing model info from analysis
- Use contracts to verify program properties
- Can detect a wide variety of issues, easy to extend

How many issues can you spot in this tiny example?

At least 8 issues in this code example!

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char** argv)
{
   int rank, size, buf[8];

   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);

   MPI_Datatype type;
   MPI_Type_contiguous (2, MPI_INTEGER, &type);

   MPI_Recv (buf, 2, MPI_INT, size - rank, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   MPI_Send (buf, 2, type, size - rank, 123, MPI_COMM_WORLD);
   printf ("Hello, I am rank %d of %d.\n", rank, size);

   return 0;
}
```

MPI + GPU Correctness Checking with MUST
46th VI-HPS Workshop
Joachim Jenke, Felix Tomski

Conclusion

Tool Summary:

- Ensure modularity by removing model info from analysis
- Use contracts to verify program properties
- Can detect a wide variety of issues, easy to extend

Evaluation Results:

- Wide coverage grants high accuracy
- Data Flow analysis comparatively cheap

How many issues can you spot in this tiny example?

At least 8 issues in this code example!

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char** argv)
{
   int rank, size, buf[8];

   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);

   MPI_Datatype type;
   MPI_Type_contiguous (2, MPI_INTEGER, &type);

   MPI_Recv (buf, 2, MPI_INT, size - rank, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   MPI_Send (buf, 2, type, size - rank, 123, MPI_COMM_WORLD);
   printf ("Hello, I am rank %d of %d.\n", rank, size);

   return 0;
}
```

MPI + GPU Correctness Checking with MUST
46th VI-HPS Workshop
Joachim Jenke, Felix Tomski

Conclusion

Tool Summary:

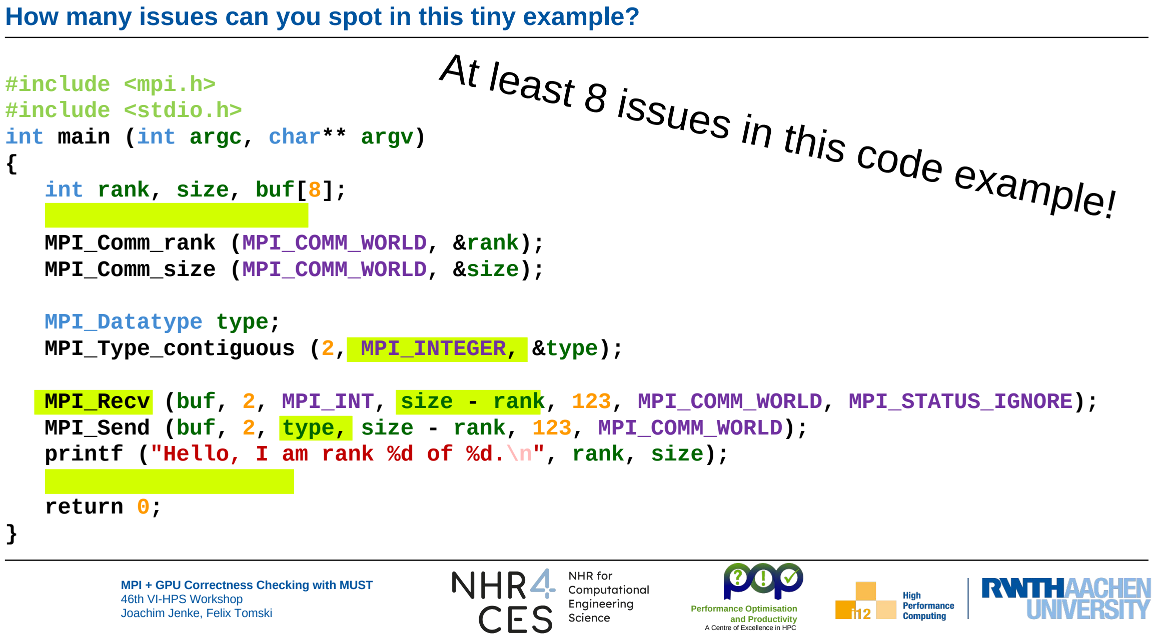
- Ensure modularity by removing model info from analysis
- Use contracts to verify program properties
- Can detect a wide variety of issues, easy to extend

Evaluation Results:

- Wide coverage grants high accuracy
- Data Flow analysis comparatively cheap

Future Work:

- Improve analysis accuracy
- Extend programming language support
- Extend contract language for further error classes



How many issues can you spot in this tiny example?

At least 8 issues in this code example!

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char** argv)
{
    int rank, size, buf[8];

    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    MPI_Datatype type;
    MPI_Type_contiguous (2, MPI_INTEGER, &type);

    MPI_Recv (buf, 2, MPI_INT, size - rank, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send (buf, 2, type, size - rank, 123, MPI_COMM_WORLD);
    printf ("Hello, I am rank %d of %d.\n", rank, size);

    return 0;
}
```

MPI + GPU Correctness Checking with MUST
46th VI-HPS Workshop
Joachim Jenke, Felix Tomski

Conclusion

Tool Summary:

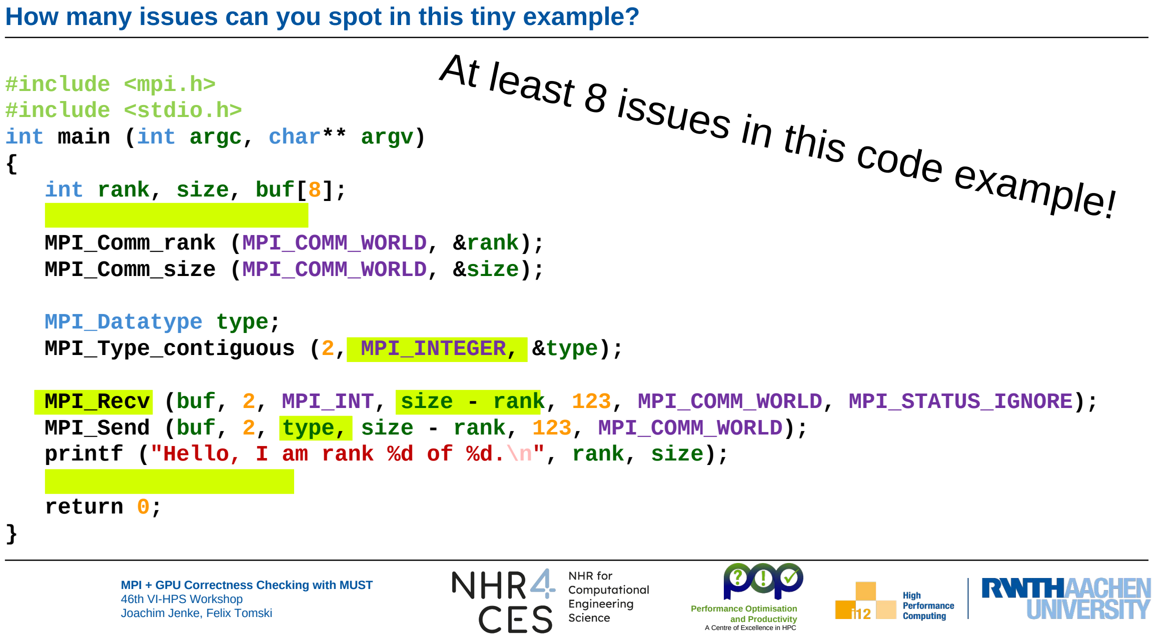
- Ensure modularity by removing model info from analysis
- Use contracts to verify program properties
- Can detect a wide variety of issues, easy to extend

Evaluation Results:

- Wide coverage grants high accuracy
- Data Flow analysis comparatively cheap

Future Work:

- Improve analysis accuracy
- Extend programming language support
- Extend contract language for further error classes


How many issues can you spot in this tiny example?

At least 8 issues in this code example!

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char** argv)
{
    int rank, size, buf[8];

    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    MPI_Datatype type;
    MPI_Type_contiguous (2, MPI_INTEGER, &type);

    MPI_Recv (buf, 2, MPI_INT, size - rank, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send (buf, 2, type, size - rank, 123, MPI_COMM_WORLD);
    printf ("Hello, I am rank %d of %d.\n", rank, size);

    return 0;
}
```

MPI + GPU Correctness Checking with MUST
46th VI-HPS Workshop
Joachim Jenke, Felix Tomski

Thank you for your attention!

References I

[1] Vytautas Astrauskas et al. “The Prusti Project: Formal Verification for Rust”. In: *NASA Formal Methods*. Cham: Springer International Publishing, 2022, pp. 88–108. ISBN: 978-3-031-06773-0. DOI: 10.1007/978-3-031-06773-0_5.

[2] Semih Burak et al. “SPMD IR: Unifying SPMD and Multi-value IR Showcased for Static Verification of Collectives”. In: *Recent Advances in the Message Passing Interface*. Cham: Springer Nature Switzerland, 2025, pp. 3–20. ISBN: 978-3-031-73370-3. DOI: 10.1007/978-3-031-73370-3_1.

[3] OpenSHMEM Committee. *OpenSHMEM: Application Programming Interface Version 1.5*. 2020. URL: http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.5.pdf (visited on 12/30/2024).

[4] Timur Doumler and Jens Maurer. *A Natural Syntax for Contracts*. Nov. 8, 2023. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2961r2.pdf.

[5] 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. New York, NY, USA: Association for Computing Machinery, Nov. 15, 2015, pp. 1–10. ISBN: 978-1-4503-4005-2. DOI: 10.1145/2833157.2833159. URL: https://dl.acm.org/doi/10.1145/2833157.2833159 (visited on 11/04/2024).

References II

[6] GASPI Forum. *GASPI: Global Address Space Programming Interface 17.1*. 2017. URL: https://raw.githubusercontent.com/GASPI-Forum/GASPI-Forum.github.io/master/standards/GASPI-17.1.pdf (visited on 12/30/2024).

[7] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*. 2023. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[8] Sayan Ghosh et al. "MiniVite: A Graph Analytics Benchmarking Tool for Massively Parallel Systems". In: *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). Nov. 2018, pp. 51–56. DOI: 10.1109/PMBS.2018.8641631. URL: https://ieeexplore.ieee.org/abstract/document/8641631 (visited on 01/19/2025).

[9] Tobias Hilbrich et al. "MUST: A Scalable Approach to Runtime Error Detection in MPI Programs". In: *Tools for High Performance Computing 2009*. Berlin, Heidelberg: Springer, 2010, pp. 53–66. ISBN: 978-3-642-11261-4. DOI: 10.1007/978-3-642-11261-4_5.

References III

[10] Tim Jammer et al. “MPI-BugBench: A Framework for Assessing MPI Correctness Tools”. In: *Recent Advances in the Message Passing Interface*. Cham: Springer Nature Switzerland, 2025, pp. 121–137. ISBN: 978-3-031-73370-3. DOI: 10.1007/978-3-031-73370-3_8.

[11] I Karlin, J Keasler, and J Neely. *LULESH 2.0 Updates and Changes*. LLNL-TR-641973, 1090032. July 22, 2013, LLNL-TR–641973, 1090032. DOI: 10.2172/1090032. URL: https://www.osti.gov/servlets/purl/1090032/ (visited on 01/19/2025).

[12] *Kotlin Contracts Proposal*. GitHub. Dec. 26, 2019. URL: https://github.com/Kotlin/KEEP/blob/master/proposals/kotlin-contracts.md (visited on 02/07/2025).

[13] Patrick Ohly and Werner Krotz-Vogel. “Automated MPI Correctness Checking What If There Was a Magic Option?” In: *Proceedings of the 8th LCI International Conference on High-Performance Clustered Computing*. May 2007, pp. 19–25.

[14] T. J. Parr and R. W. Quong. “ANTLR: A Predicated-LL(k) Parser Generator”. In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810. ISSN: 1097-024X. DOI: 10.1002/spe.4380250705. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380250705 (visited on 11/30/2024).

References IV

[15] Emmanuelle Saillard et al. "Static Local Concurrency Errors Detection in MPI-RMA Programs". In: *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)*. 2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness). Nov. 2022, pp. 18–26. DOI: `10.1109/Correctness56720.2022.00008`. URL: `https://ieeexplore.ieee.org/abstract/document/10027512` (visited on 11/04/2024).

[16] Simon Schwitanski et al. "RMARaceBench: A Microbenchmark Suite to Evaluate Race Detection Tools for RMA Programs". In: *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W '23. New York, NY, USA: Association for Computing Machinery, Nov. 12, 2023, pp. 205–214. ISBN: 979-8-4007-0785-8. DOI: `10.1145/3624062.3624087`. URL: `https://dl.acm.org/doi/10.1145/3624062.3624087` (visited on 12/08/2024).

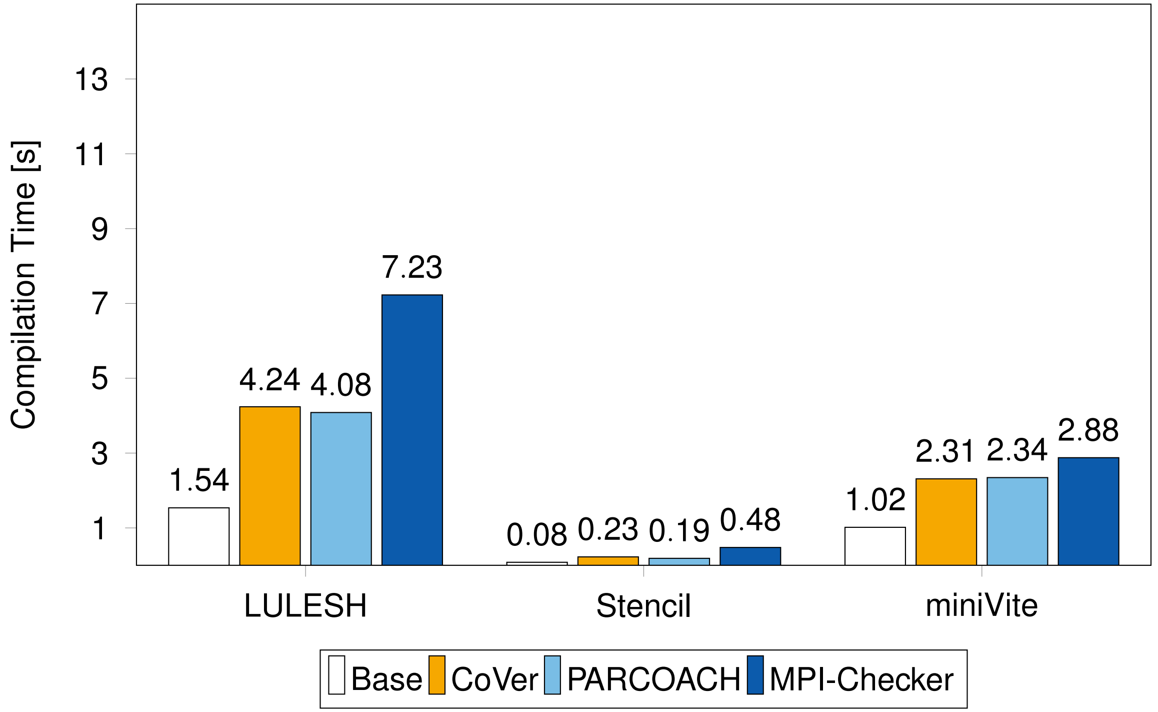
[17] Rob F. Van der Wijngaart and Timothy G. Mattson. "The Parallel Research Kernels". In: *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. 2014 IEEE High Performance Extreme Computing Conference (HPEC). Sept. 2014, pp. 1–6. DOI: `10.1109/HPEC.2014.7040972`. URL: `https://ieeexplore.ieee.org/abstract/document/7040972` (visited on 01/19/2025).

Common False-Positive Causes

```
if (rank == 0) {
    MPI_Irecv(buf, ..., MPI_REQUEST_NULL);
}
if (rank == 1) {
    MPI_Isend(buf, ..., &mpi_request_0);
}
MPI_Wait(&mpi_request_0, MPI_STATUS_IGNORE);
```

- Branch conditions not checked
  - Future Work: Evaluate Multi-Value analysis for improvement
- Path with both conditions evaluated to true checked
- $\implies$ Invalid data race report
- Duplicate error: Unmatched Wait, Missing Wait

Absolute Compilation Time Comparison



Output from introduction example

```
## Contract violation detected! ##
--> Function: MPI_Comm_rank
--> Contract: ...
    --> Message: "Missing Initialization call"
...
    --> Message: "Missing Finalization call"
// Init-/Finalization errors left out for other functions

## Contract violation detected! ##
--> Function: MPI_Type_contiguous
--> Contract: ...
    --> Message: "Type not committed before use"
```