



Allocation Tracking and Parameter Checking for Parallel Programming Models using Contracts

Yussur Mustafa Orajı



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Parallel Programming

- Working with language-native parallel programming difficult
 - Different language → different API
 - Low-level, little abstraction
- In contrast: Parallel Programming Models
 - MPI [1]
 - OpenSHMEM [2]
 - GASPI [3]
 - ...
- (Sometimes) support for multiple languages
- Very error-prone...

Why Contracts?

- Current example: MPI, but OpenSHMEM and other APIs are used as well
- Main languages: C/C++ and Fortran
- Instead of one tool per programming model $\times$ language: Generic tooling
- Verification rules not set by tool, but using contracts

Why Contracts?

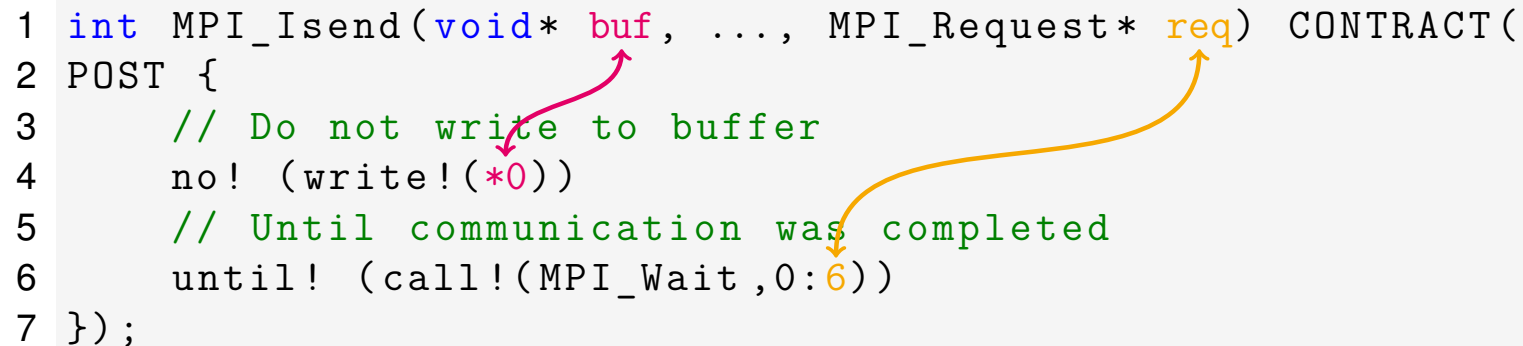
- Current example: MPI, but OpenSHMEM and other APIs are used as well
- Main languages: C/C++ and Fortran
- Instead of one tool per programming model $\times$ language: Generic tooling
- Verification rules not set by tool, but using contracts $\rightarrow$ Example for CoVer:

```
1 int MPI_Isend(void* buf, ..., MPI_Request* req) CONTRACT(  
2 POST {  
3     // Do not write to buffer  
4     no! (write!(*0))  
5     // Until communication was completed  
6     until! (call!(MPI_Wait,0:6))  
7 });
```

Why Contracts?

- Current example: MPI, but OpenSHMEM and other APIs are used as well
- Main languages: C/C++ and Fortran
- Instead of one tool per programming model $\times$ language: Generic tooling
- Verification rules not set by tool, but using contracts $\rightarrow$ Example for CoVer:

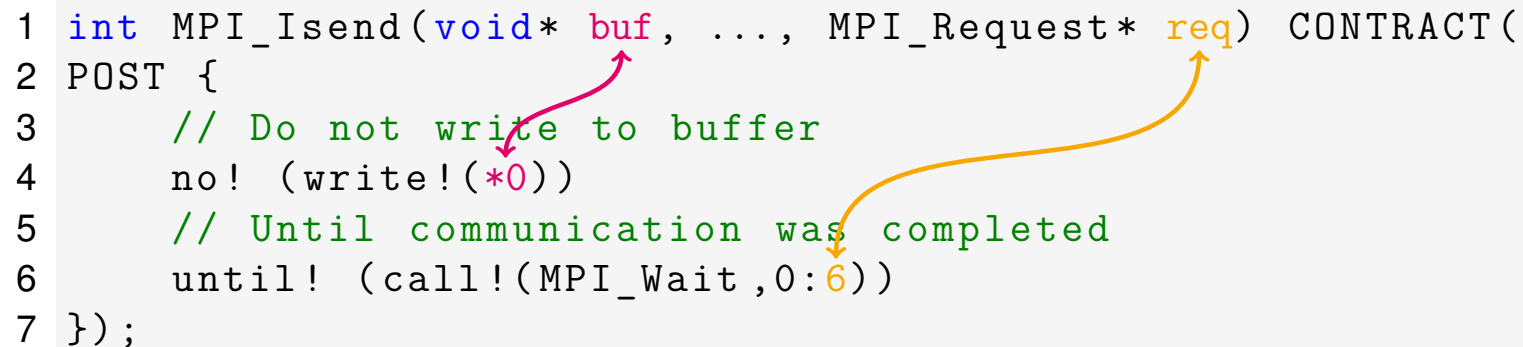
```
1 int MPI_Isend(void* buf, ..., MPI_Request* req) CONTRACT(  
2 POST {  
3     // Do not write to buffer  
4     no! (write!(*0))  
5     // Until communication was completed  
6     until! (call!(MPI_Wait,0:6))  
7 });
```



Why Contracts?

- Current example: MPI, but OpenSHMEM and other APIs are used as well
- Main languages: C/C++ and Fortran
- Instead of one tool per programming model $\times$ language: Generic tooling
- Verification rules not set by tool, but using contracts $\rightarrow$ Example for CoVer:

```
1 int MPI_Isend(void* buf, ..., MPI_Request* req) CONTRACT(  
2 POST {  
3     // Do not write to buffer  
4     no! (write!(*0))  
5     // Until communication was completed  
6     until! (call!(MPI_Wait,0:6))  
7 });
```

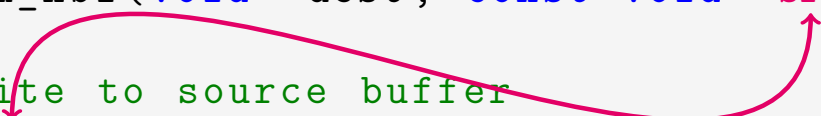


Analyses unaware of model: No need for additional analyses for other models!

Why Contracts?

- Current example: MPI, but OpenSHMEM and other APIs are used as well
- Main languages: C/C++ and Fortran
- Instead of one tool per programming model $\times$ language: Generic tooling
- Verification rules not set by tool, but using contracts $\rightarrow$ Example for CoVer:

```
1 void shmem_putmem_nbi(void* dest, const void* src, ...) CONTRACT(  
2 POST {  
3     // Do not write to source buffer  
4     no! (write!(*1))  
5     // Until communication was completed  
6     until! (call!(shmem_quiet))  
7 });
```

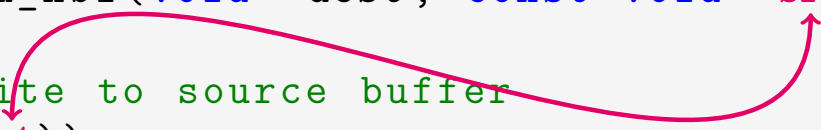


Analyses unaware of model: No need for additional analyses for other models!

Why Contracts?

- Current example: MPI, but OpenSHMEM and other APIs are used as well
- Main languages: C/C++ and Fortran
- Instead of one tool per programming model $\times$ language: Generic tooling
- Verification rules not set by tool, but using contracts $\rightarrow$ Example for CoVer:

```
1 void shmem_putmem_nbi(void* dest, const void* src, ...) CONTRACT(  
2 POST {  
3     // Do not write to source buffer  
4     no! (write!(*1))  
5     // Until communication was completed  
6     until! (call!(shmem_quiet))  
7 });
```



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

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

Error Classes

- Contracts written as formulas of basic operations
- Examples:
 - $\text{MPI_} * \rightarrow \text{PRE} \{ \text{call!}(\text{MPI_Init}) \}$
 - $\text{MPI_} * \rightarrow \text{POST} \{ \text{call!}(\text{MPI_Finalize}) \}$
 - $\text{MPI_Isend} \rightarrow \text{POST} \{ \text{no! } (\text{read!}(\text{buf})) \text{ until } (\text{call!}(\text{MPI_Wait})) \}$
- Constructs can be refined (e.g. respect request handles, recognize `MPI_Init_thread, ...`)
- Many error classes representable:
 - Missing Initialization/Finalization
 - Data Races

Error Classes

- Contracts written as formulas of basic operations
- Examples:
 - $\text{MPI_} * \rightarrow \text{PRE } \{ \text{call!}(\text{MPI_Init}) \}$
 - $\text{MPI_} * \rightarrow \text{POST } \{ \text{call!}(\text{MPI_Finalize}) \}$
 - $\text{MPI_Isend} \rightarrow \text{POST } \{ \text{no! } (\text{read!}(\text{buf})) \text{ until } (\text{call!}(\text{MPI_Wait})) \}$
- Constructs can be refined (e.g. respect request handles, recognize `MPI_Init_thread, ...`)
- Many error classes representable:
 - Missing Initialization/Finalization
 - Data Races
 - Request Leaks ($\text{MPI_Isend} \rightarrow \text{POST } \{ \text{call!}(\text{MPI_Wait}) \}$)
 - Uncommitted Datatype ($\text{MPI_Type_vector} \rightarrow \text{POST } \{ \text{no! } (\text{call!}(\text{MPI_Send})) \text{ until! } (\text{call!}(\text{MPI_Type_commit})) \}$)
 - Missing or Mixed Epoch ($\text{MPI_Get} \rightarrow \text{PRE } \{ \text{call!}(\text{MPI_Win_fence}) \wedge \text{call!}(\text{MPI_Win_lock}) \wedge \text{call!}(\text{MPI_Win_start}) \}$)
 - ...

Error Classes

- Current implementation limited, does not recognize parameter issues

Error Classes

- Current implementation limited, does not recognize parameter issues
- Possible missed errors:
 - Invalid Parameters

```
1 int* buf = (int*)malloc(sizeof(int));  
2 MPI_Send(buf, ..., MPI_ANY_SOURCE); // Cannot send to MPI_ANY_SOURCE!  
3 MPI_Recv(buf, ..., -1); // Invalid Rank!
```

Error Classes

- Current implementation limited, does not recognize parameter issues
- Possible missed errors:
 - Invalid Parameters
 - Use of unallocated buffer (NULL or uninitialized)

```
1 int* buf;  
2 MPI_Send(buf, ..., 1); // buf is uninitialized!  
3 MPI_Recv(NULL, ..., 1); // NULL is not a valid buffer!
```

Error Classes

- Current implementation limited, does not recognize parameter issues
- Possible missed errors:
 - Invalid Parameters
 - Use of unallocated buffer (NULL or uninitialized)
 - Use-after-free

```
1 int* buf = (int*)malloc(sizeof(int));  
2 free(buf);  
3 MPI_Send(buf, ..., 1); // buf is no longer allocated!
```

Error Classes

- Current implementation limited, does not recognize parameter issues
- Possible missed errors:
 - Invalid Parameters
 - Use of unallocated buffer (NULL or uninitialized)
 - Use-after-free

- Errors cannot be described by current contract language

⇒ Extension first needs new contract semantics

Language Extension

- New param operation: Compare and verify values

```
1 int MPI_Recv(void *buf, int count, ..., int source, int tag, ...) CONTRACT(  
2     PRE {  
3         param!(3: >= 0), // Rank must be non-negative  
4         param!(4: >= 0), // Tag must be non-negative  
5         param!(0: != NULL, != MPI_IN_PLACE) // Buffer is not one of these constants  
6     });
```

Language Extension

- New param operation: Compare and verify values

```
1 int MPI_Recv(void *buf, int count, ..., int source, int tag, ...) CONTRACT(  
2     PRE {  
3         param!(3: >= 0), // Rank must be non-negative  
4         param!(4: >= 0), // Tag must be non-negative  
5         param!(0: != NULL, != MPI_IN_PLACE) // Buffer is not one of these constants  
6     });
```

- **Insufficient:** Magic Numbers can cause conflicts with “intended” semantics
 - Example: OpenMPI defines MPI_ANY_SOURCE internally as -1
- ⇒ Clashes with established contract, false positives!

Language Extension

- New param operation: Compare and verify values

```
1 int MPI_Recv(void *buf, int count, ..., int source, int tag, ...) CONTRACT(  
2     PRE {  
3         param!(3:~=MPI_ANY_SOURCE, >=0), // Rank must be non-negative  
4         param!(4:>=0), // Tag must be non-negative  
5         param!(0:!=NULL, !=MPI_IN_PLACE) // Buffer is not one of these constants  
6     });
```

- Use exceptions to permit magic values where needed
 - Example: OpenMPI defines MPI_ANY_SOURCE internally as -1
- ⇒ Allowed by exception, false positive avoided

Language Extension

- New param operation: Compare and verify values

```
1 int MPI_Recv(void *buf, int count, ..., int source, int tag, ...) CONTRACT(  
2     PRE {  
3         param!(3:~=MPI_ANY_SOURCE, >=0), // Rank must be non-negative  
4         param!(4:>=0), // Tag must be non-negative  
5         param!(0:!=NULL, !=MPI_IN_PLACE) // Buffer is not one of these constants  
6     });
```

- Use exceptions to permit magic values where needed
 - Example: OpenMPI defines MPI_ANY_SOURCE internally as -1
- ⇒ Allowed by exception, false positive avoided
- **Insufficient:** Macro values unknown to CoVer, cannot reference in contract
- ⇒ Macros expanded prior to compile-time pass, cannot resolve name and contract is ill-formed!

Language Extension

- New param operation: Compare and verify values

```
1 int MPI_Recv(void *buf, int count, ..., int source, int tag, ...) CONTRACT(  
2     PRE {  
3         param!(3:~=MPI_ANY_SOURCE, >=0), // Rank must be non-negative  
4         param!(4:>=0), // Tag must be non-negative  
5         param!(0:!=NULL, !=MPI_IN_PLACE) // Buffer is not one of these constants  
6     });
```

- Use exceptions to permit magic values where needed
 - Example: OpenMPI defines MPI_ANY_SOURCE internally as -1
- ⇒ Allowed by exception, false positive avoided
- **Insufficient:** Macro values unknown to CoVer, cannot reference in contract
- ⇒ Declare macro names ahead of time, forward to analysis

```
1 COVER_VALUE_PAIR(MPI_ANY_SOURCE, MPI_ANY_SOURCE)
```

Static Analysis

- First: Need to parse value declarations as well
 - Map string name to expanded macro values
- For each `param` operation, perform analysis at each callsite
- At each callsite, backtrack to infer value and check against requirement

```
1 int* buf = NULL;
2 if (rank == 0) {
3     buf = (int*)malloc(sizeof(int)); *buf = 42;
4     MPI_Send(buf, ...);
5 } else {
6     MPI_Recv(buf, ...); // buf is NULL!
7 }
```

Static Analysis

- First: Need to parse value declarations as well
 - Map string name to expanded macro values
- For each `param` operation, perform analysis at each callsite
- At each callsite, backtrack to infer value and check against requirement

```
1 int* buf = NULL;
2 if (rank == 0) {
3     buf = (int*)malloc(sizeof(int)); *buf = 42;
4     MPI_Send(buf, ...);
5 } else {
6     MPI_Recv(buf, ...); // buf is NULL!
7 }
```

- **But:** Backtracking is fragile, and requires many heuristics to function, especially on Fortran
- Improve analysis stability by delaying optimization past analysis

Language Extension

- Two new operations, `alloc` and `free`
- `alloc`:
 - PRE: Parameter must be allocated prior to calling function
 - POST: Parameter will be allocated after calling function

1
2
3
4
5
6
7
8

Language Extension

- Two new operations, alloc and free
- alloc:
 - PRE: Parameter must be allocated prior to calling function
 - POST: Parameter will be allocated after calling function

```
1 int MPI_Send(void* buf, ...) CONTRACT(  
2     PRE { alloc!(0) }); // First argument must be allocated  
3  
4  
5  
6  
7  
8
```

Language Extension

- Two new operations, alloc and free
- alloc:
 - PRE: Parameter must be allocated prior to calling function
 - POST: Parameter will be allocated after calling function

```
1 int MPI_Send(void* buf, ...) CONTRACT(  
2     PRE { alloc!(0) }); // First argument must be allocated  
3 void* malloc(size_t size) CONTRACT(  
4     POST { alloc!(R[0 _arg]) }); // Allocates return value, size in first argument  
5 void* calloc(size_t num, size_t size) CONTRACT(  
6     POST { alloc!(R[0 _arg * 1 _arg]) }); // Total size: num *element size  
7  
8
```

Language Extension

- Two new operations, alloc and free
- alloc:
 - PRE: Parameter must be allocated prior to calling function
 - POST: Parameter will be allocated after calling function

```
1 int MPI_Send(void* buf, ...) CONTRACT(  
2     PRE { alloc!(0) }); // First argument must be allocated  
3 void* malloc(size_t size) CONTRACT(  
4     POST { alloc!(R[0 _arg]) }); // Allocates return value, size in first argument  
5 void* calloc(size_t num, size_t size) CONTRACT(  
6     POST { alloc!(R[0 _arg * 1 _arg]) }); // Total size: num * element size  
7 int MPI_Win_allocate(MPI_Aint size, ..., void* baseptr, ...) CONTRACT(  
8     POST{ alloc!(*4[0 _arg]) }); // Allocates pointer in fifth argument, size in first argument
```

Language Extension

- Two new operations, `alloc` and `free`
- `alloc`:
 - PRE: Parameter must be allocated prior to calling function
 - POST: Parameter will be allocated after calling function
- `free`: Only in POST, parameter freed after calling function

```
1 void free(void* ptr) CONTRACT(  
2     POST { free!(0) }); // Free pointer in first argument
```

Static Analysis

- Forward data-flow analysis collects allocations
- Create allocation graph (new allocations and copies)
 - Add new allocation for callsites with `alloc` in POST
 - If value is copied: Add node for copy and edge from original
 - Remove full tree if value is freed (`free` in POST)
- For each `alloc` in PRE: Check if value in allocation graph

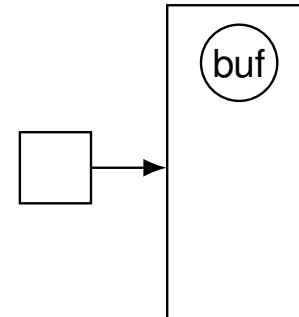
```
1 int* buf = malloc(sizeof(int));
2 int* y = buf;
3 int* z = buf;
4 int* y1 = y;
5 MPI_Send(y1 ...);
6 free(y1);
7 MPI_Send(buf, ...);
```



Static Analysis

- Forward data-flow analysis collects allocations
- Create allocation graph (new allocations and copies)
 - Add new allocation for callsites with `alloc` in POST
 - If value is copied: Add node for copy and edge from original
 - Remove full tree if value is freed (`free` in POST)
- For each `alloc` in PRE: Check if value in allocation graph

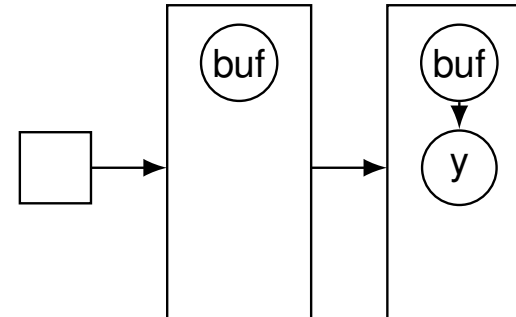
```
1 int* buf = malloc(sizeof(int));
2 int* y = buf;
3 int* z = buf;
4 int* y1 = y;
5 MPI_Send(y1 ...);
6 free(y1);
7 MPI_Send(buf, ...);
```



Static Analysis

- Forward data-flow analysis collects allocations
- Create allocation graph (new allocations and copies)
 - Add new allocation for callsites with `alloc` in POST
 - If value is copied: Add node for copy and edge from original
 - Remove full tree if value is freed (`free` in POST)
- For each `alloc` in PRE: Check if value in allocation graph

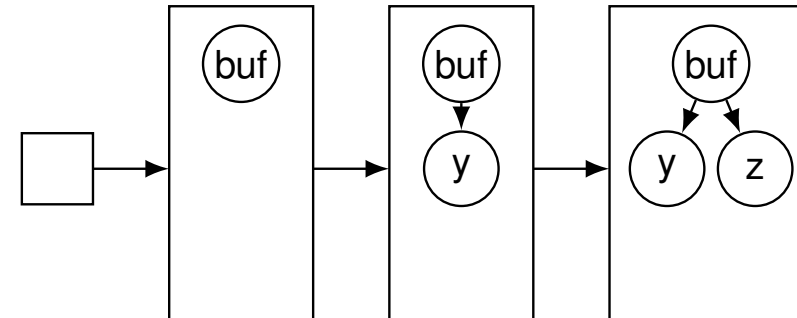
```
1 int* buf = malloc(sizeof(int));
2 int* y = buf;
3 int* z = buf;
4 int* y1 = y;
5 MPI_Send(y1 ...);
6 free(y1);
7 MPI_Send(buf, ...);
```



Static Analysis

- Forward data-flow analysis collects allocations
- Create allocation graph (new allocations and copies)
 - Add new allocation for callsites with `alloc` in POST
 - If value is copied: Add node for copy and edge from original
 - Remove full tree if value is freed (`free` in POST)
- For each `alloc` in PRE: Check if value in allocation graph

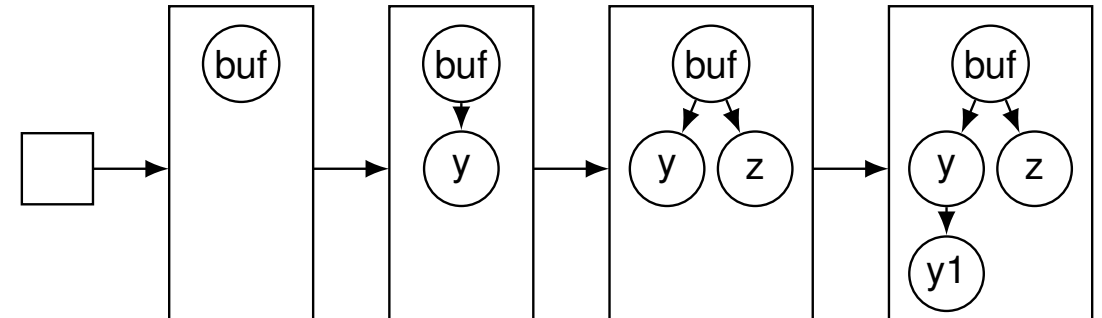
```
1 int* buf = malloc(sizeof(int));
2 int* y = buf;
3 int* z = buf;
4 int* y1 = y;
5 MPI_Send(y1 ...);
6 free(y1);
7 MPI_Send(buf, ...);
```



Static Analysis

- Forward data-flow analysis collects allocations
- Create allocation graph (new allocations and copies)
 - Add new allocation for callsites with `alloc` in POST
 - If value is copied: Add node for copy and edge from original
 - Remove full tree if value is freed (`free` in POST)
- For each `alloc` in PRE: Check if value in allocation graph

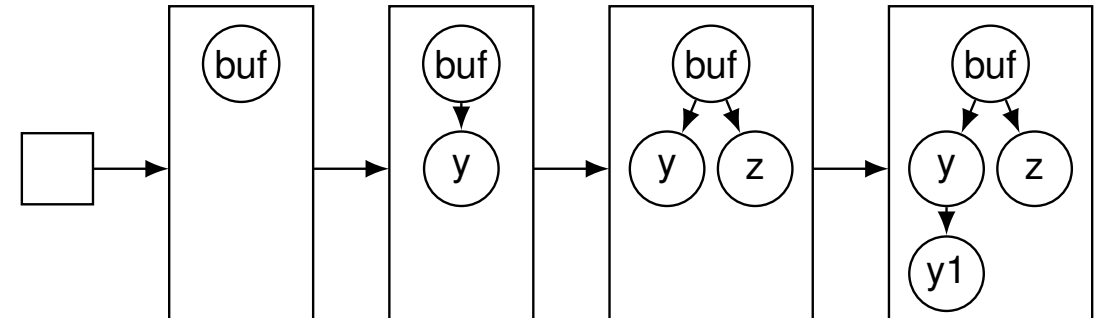
```
1 int* buf = malloc(sizeof(int));
2 int* y = buf;
3 int* z = buf;
4 int* y1 = y;
5 MPI_Send(y1 ...);
6 free(y1);
7 MPI_Send(buf, ...);
```



Static Analysis

- Forward data-flow analysis collects allocations
- Create allocation graph (new allocations and copies)
 - Add new allocation for callsites with `alloc` in POST
 - If value is copied: Add node for copy and edge from original
 - Remove full tree if value is freed (`free` in POST)
- For each `alloc` in PRE: Check if value in allocation graph

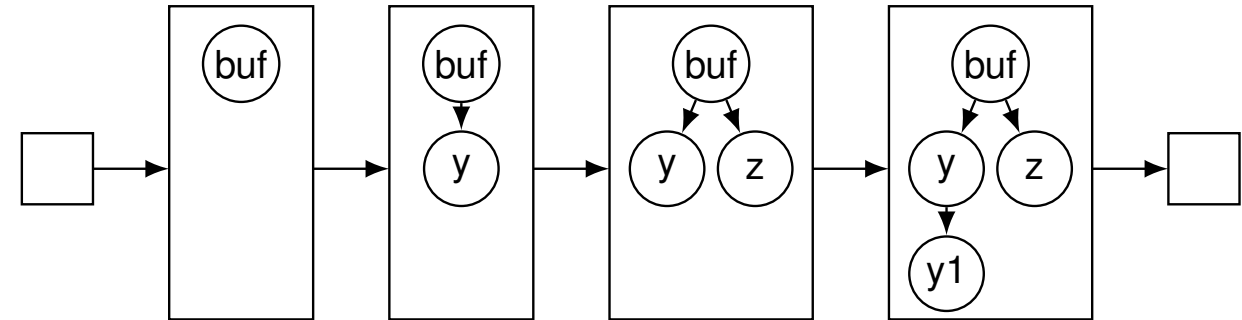
```
1 int* buf = malloc(sizeof(int));
2 int* y = buf;
3 int* z = buf;
4 int* y1 = y;
5 MPI_Send(y1 ...);
6 free(y1);
7 MPI_Send(buf, ...);
```



Static Analysis

- Forward data-flow analysis collects allocations
- Create allocation graph (new allocations and copies)
 - Add new allocation for callsites with `alloc` in POST
 - If value is copied: Add node for copy and edge from original
 - Remove full tree if value is freed (`free` in POST)
- For each `alloc` in PRE: Check if value in allocation graph

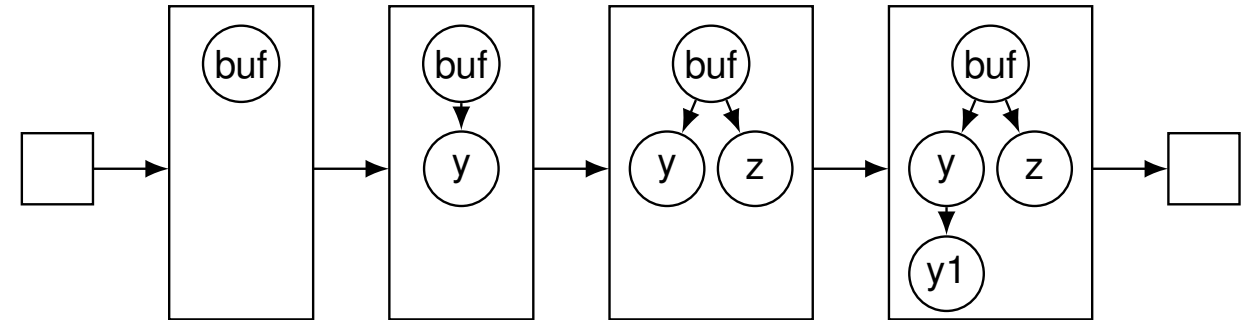
```
1 int* buf = malloc(sizeof(int));
2 int* y = buf;
3 int* z = buf;
4 int* y1 = y;
5 MPI_Send(y1 ...);
6 free(y1);
7 MPI_Send(buf, ...);
```



Static Analysis

- Forward data-flow analysis collects allocations
- Create allocation graph (new allocations and copies)
 - Add new allocation for callsites with `alloc` in POST
 - If value is copied: Add node for copy and edge from original
 - Remove full tree if value is freed (`free` in POST)
- For each `alloc` in PRE: Check if value in allocation graph

```
1 int* buf = malloc(sizeof(int));
2 int* y = buf;
3 int* z = buf;
4 int* y1 = y;
5 MPI_Send(y1 ...);
6 free(y1);
7 MPI_Send(buf, ...); // Use after free!
```



Dynamic Analysis

- Current dynamic analysis infrastructure insufficient for allocation tracking
- Callbacks inserted prior to function call

```
1 int* buf;  
2 CoVer_FunctionCB(MPI_Win_allocate, ..., &buf, ...);  
3 MPI_Win_allocate(..., &buf, ...);
```

- **Cannot intercept and store allocated pointer value!**
 - Cannot use PMPI: MPI specific, cannot use across programming models
 - Cannot add another callback: Slowdown too severe

Dynamic Analysis

- Current dynamic analysis infrastructure insufficient for allocation tracking
- Callbacks inserted prior to function call

```
1 int* buf;  
2 CoVer_FunctionCB(MPI_Win_allocate, ..., &buf, ...);  
3 // Original function call deleted
```

```
1 ... // Analyses before function call  
2 ffi_call(MPI_Win_allocate, ...)  
3 ... // Analyses after function call
```

- Use libffi to create “our own” PMPI
 - libffi: Foreign Function Interface library
- Create dynamic calls at runtime using libffi
 - Same ABI → little overhead
 - Function Pointer and arguments forwarded

- Allocation tracking works... sometimes

```
1 void* malloc(size_t size)
2     CONTRACT( POST { alloc!(R[0 _arg]) });
3
4 int main() {
5     ...
6     int* buf;
7     buf = (int*)malloc(sizeof(int));
8     MPI_Send(buf, ...);
9     ...
10 }
```

Intrinsics

- Allocation tracking works... sometimes

```
1 void* malloc(size_t size)
2     CONTRACT( POST { alloc!(R[0 _arg]) } );
3
4 int main() {
5     ...
6     int* buf;
7     buf = (int*)malloc(sizeof(int));
8     MPI_Send(buf, ...);
9     ...
10 }
```

```
1
2
3
4 int main() {
5     ...
6     int buf;
7
8     MPI_Send(&buf, ...);
9     ...
10 }
```

- **No function call to intercept for stack variables!**
 - Generally: Language-specific allocations, including stack variables, globals and Fortran allocate

Intrinsics

- Allocation tracking works... sometimes

```
1 void* malloc(size_t size)
2     CONTRACT( POST { alloc!(R[0 _arg]) });
3
4 int main() {
5     ...
6     int* buf;
7     buf = (int*)malloc(sizeof(int));
8     MPI_Send(buf, ...);
9     ...
10 }
```

```
1 void CoVer_AllocStack(void* ptr, size_t size)
2     CONTRACT( POST { alloc!(0[1 _arg]) });
3
4 int main() {
5     ...
6     int buf;
7     CoVer_AllocStack(&buf, sizeof(int));
8     MPI_Send(&buf, ...);
9     ...
10 }
```

- Add intrinsic functions to capture language semantics
 - Generally: Language-specific allocations, including stack variables, globals and Fortran allocate
- Deallocation callbacks also added at end of function call

Evaluation Setup

- Evaluation performed for overhead and classification quality
- Classification Quality:
 - MPI-BugBench [4] level 1
 - MPI-BugBench Fortran [5]
- Overhead:
 - PRK Stencil [6] Kernel
 - miniWeather [7] Proxy App
 - LULESH [8] Proxy App
- All tests performed on the Lichtenberg II cluster of TU Darmstadt
- Per node, connected via InfiniBand:
 - 2x Intel Xeon Platinum 9232 CPUs (48 cores each)
 - 384GB RAM
- Compare against tool MUST [9]

Classification Quality - Supported Error Classes

Tool	C				Fortran				Difference			
	TP	TN	FN	Acc	TP	TN	FN	Acc	TP	TN	FN	Acc
CoVer-Dynamic	49	23	3	0.96	49	23	3	0.96	+0	+0	+0	+0.00
CoVer-Static	48	23	4	0.95	48	23	4	0.95	+0	+0	+0	+0.00
CoVer-Dynamic (Old)	27	23	25	0.67	27	23	25	0.67	+0	+0	+0	+0.00
CoVer-Static (Old)	27	23	25	0.67	27	23	25	0.67	+0	+0	+0	+0.00
MUST	43	23	9	0.88	21	23	31	0.59	-22	+0	+22	-0.29

- CoVer: High Accuracy across C and Fortran
- Significant improvements to TP rate
 - High concentration of parameter error cases
- MUST: Missing TypeART on Fortran → No support for allocation tracking
- **But:** Unsupported testcases (deadlocks, remote races, data type mismatch) excluded
 - Future Work

Classification Quality - Implementation Differences

- Sometimes, different results depending on MPI implementation
- Contract: PRE { param! (3: ^=MPI_PROC_NULL, >=0) }

```
1 // OpenMPI
2
3
4 MPI_Send(..., /* rank */ -1, ...);
```

Error Detected!

```
1 // MPICH
2
3
4 MPI_Send(..., /* rank */ -1, ...);
```

False Negative!

Classification Quality - Implementation Differences

- Sometimes, different results depending on MPI implementation
- Contract: PRE { param! (3: ^=MPI_PROC_NULL, >=0) }

```
1 // OpenMPI mpi.h
2 #define MPI_PROC_NULL -2
3
4 MPI_Send(..., /* rank */ -1, ...);
```

Error Detected!

- -1 does not match any exception
- -1 does not satisfy condition

→ Report Error

```
1 // MPICH mpi.h
2 #define MPI_PROC_NULL -1
3
4 MPI_Send(..., /* rank */ -1, ...);
```

False Negative!

- -1 matches exception value
- Contract is fulfilled

→ Error missed!

Classification Quality - Implementation Differences

- Sometimes, different results depending on MPI implementation
- Contract: PRE { param! (3: ^=MPI_PROC_NULL, >=0) }

```
1 // OpenMPI mpi.h
2 #define MPI_PROC_NULL -2
3
4 MPI_Send(..., /* rank */ -1, ...);
```

Error Detected!

- -1 does not match any exception
- -1 does not satisfy condition

→ Report Error

```
1 // MPICH mpi.h
2 #define MPI_PROC_NULL -1
3
4 MPI_Send(..., /* rank */ -1, ...);
```

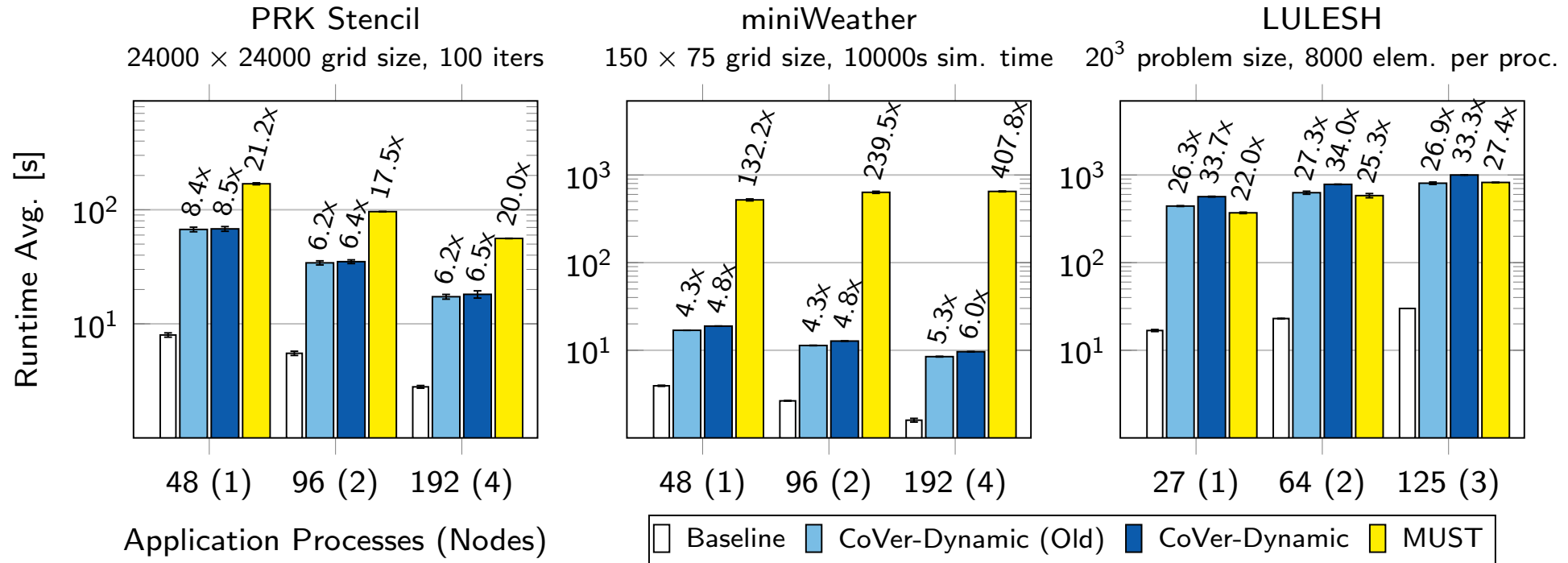
False Negative!

- -1 matches exception value
- Contract is fulfilled

→ Error missed!

Best for correctness checking: “Unlikely” Magic Values

Overhead



- CoVer: Lower overhead than MUST in most cases
- New additions cause additional overhead
 - Mostly: Stack allocation interception

Conclusion

Contributions:

- `param`: Check parameter values, exceptions
- `alloc / free`: Track allocations statically and dynamically
- Intrinsic capture language-specific information (stack, globals, Fortran `allocate`)

Conclusion

Contributions:

- `param`: Check parameter values, exceptions
- `alloc / free`: Track allocations statically and dynamically
- Intrinsic capture language-specific information (stack, globals, Fortran `allocate`)

Results:

- MPI-BugBench accuracy: 0.67 $\rightarrow$ 0.96 (dynamic) and 0.95 (static)
- Feature parity between C and Fortran
- But: Runtime stack allocation interception costs on top of previous overhead

Conclusion

Contributions:

- `param`: Check parameter values, exceptions
- `alloc / free`: Track allocations statically and dynamically
- Intrinsic capture language-specific information (stack, globals, Fortran `allocate`)

Results:

- MPI-BugBench accuracy: 0.67 $\rightarrow$ 0.96 (dynamic) and 0.95 (static)
- Feature parity between C and Fortran
- But: Runtime stack allocation interception costs on top of previous overhead

Limitations & Future Work:

- Magic values may coincide with error values depending on implementation
- Reduce overhead of stack allocation tracking
- Support additional error classes, e.g. deadlocks

Conclusion

Contributions:

- `param`: Check parameter values, exceptions
- `alloc / free`: Track allocations statically and dynamically
- Ininsics capture language-specific information (stack, globals, Fortran `allocate`)

Results:

- MPI-BugBench accuracy: 0.67 $\rightarrow$ 0.96 (dynamic) and 0.95 (static)
- Feature parity between C and Fortran
- But: Runtime stack allocation interception costs on top of previous overhead

Limitations & Future Work:

- Magic values may coincide with error values depending on implementation
- Reduce overhead of stack allocation tracking
- Support additional error classes, e.g. deadlocks

Thank you for your attention!

References I

- [1] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 5.0*. 2025. URL: <https://www.mpi-forum.org/docs/mpi-5.0/mpi50-report.pdf>.
- [2] 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).
- [3] 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).
- [4] Tim Jammer et al. “MPI-BugBench: A Framework for Assessing MPI Correctness Tools”. In: *Recent Advances in the Message Passing Interface*. Ed. by Claudia Blaas-Schenner, Christoph Niethammer, and Tobias Haas. Cham: Springer Nature Switzerland, 2025, pp. 121–137. ISBN: 978-3-031-73370-3. DOI: 10.1007/978-3-031-73370-3_8.

References II

- [5] Yussur Mustafa Orajı, Alexander Hück, and Christian Bischof. “Extending MPI Correctness Benchmarking to the Fortran Language”. In: *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC Workshops '25. New York, NY, USA: Association for Computing Machinery, Nov. 15, 2025, pp. 244–248. ISBN: 979-8-4007-1871-7. DOI: 10.1145/3731599.3767366.
- [6] 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.
- [7] Matthew R. Norman. *miniWeather*. Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States), Mar. 15, 2020. DOI: 10.11578/dc.20201001.88.
- [8] 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.
- [9] Tobias Hilbrich et al. “MUST: A Scalable Approach to Runtime Error Detection in MPI Programs”. In: *Tools for High Performance Computing 2009*. Ed. by Matthias S. Müller et al. Berlin, Heidelberg: Springer, 2010, pp. 53–66. ISBN: 978-3-642-11261-4. DOI: 10.1007/978-3-642-11261-4_5.