Simuliris

A Separation Logic Framework for Verifying Concurrent Program Optimizations

Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung,
Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, Derek Dreyer

MAX PLANCK INSTITUTE FOR SOFTWARE SYSTEMS
Radboud Universiteit
KAIST
SIC Saarland Informatics Campus

Clang optimizes

```
int mult(int *x, int *y) {


  int i = 0; int sum = *y;
  while (i != *x - 1) {
    i += 1; sum += *y;
  }
  return sum;
}
```

```
int opt(int *x, int *y) {
  int n = *x;
  int m = *y;
  int i = 0; int sum = m;
  while (i != n - 1) {
      i += 1; sum += m;
  }
  return sum;
}
```

implements multiplication of positive integers

Clang optimizes

```
int mult(int *x, int *y) {


  int i = 0; int sum = *y;
  while (i != *x - 1) {
    i += 1; sum += *y;
  }
  return sum;
}
```

```
int opt(int *x, int *y) {
  int n = *x;
  int m = *y;
  int i = 0; int sum = m;
  while (i != n - 1) {
      i += 1; sum += m;
  }
  return sum;
}
```

implements multiplication of positive integers

How can we prove correctness of these program optimizations?

Can concurrent writes break the optimization?

unoptimized:

```
int i = 0; int sum = *y;
while (i != *x - 1) {
  i += 1; sum += *y;
}
return sum;
```

optimized:

```
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

Can concurrent writes break the optimization?

unoptimized:

```
int i = 0; int sum = *y;
while (i != *x - 1) {
  i += 1; sum += *y;
}
return sum;
```

optimized:

```
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

||

```
*x = 2;
*y = 42;
```

Can concurrent writes break the optimization?

unoptimized:

```
int i = 0; int sum = *y;
while (i != *x - 1) {
  i += 1; sum += *y;
}
return sum;
```

optimized:

```
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

||

```
*x = 2;
*y = 42;
```

can produce results not possible for the unoptimized program!

Can concurrent writes break the optimization?

current state:

*x	*y
1	99

optimized program:

```
→int n = *x;
 int m = *y;
 int i = 0; int sum = m;
 while (i != n - 1) {
   i += 1; sum += m;
 }
 return sum;
```

||

```
→*x = 2;
 *y = 42;
```

Can concurrent writes break the optimization?

current state:

*x	*y	n
1	99	1

optimized program:

```
  int n = *x;
→int m = *y;
  int i = 0; int sum = m;
  while (i != n - 1) {
    i += 1; sum += m;
  }
  return sum;
```

||

```
→*x = 2;
  *y = 42;
```

Can concurrent writes break the optimization?

current state:

*x	*y	n
2	99	1

optimized program:

```
  int n = *x;
→int m = *y;
  int i = 0; int sum = m;
  while (i != n - 1) {
    i += 1; sum += m;
  }
  return sum;
```

||

```
  *x = 2;
→*y = 42;
```

Can concurrent writes break the optimization?

current state:

*x	*y	n
2	42	1

optimized program:

```
  int n = *x;
→ int m = *y;
  int i = 0; int sum = m;
  while (i != n - 1) {
    i += 1; sum += m;
  }
  return sum;
```

||

```
  *x = 2;
  *y = 42;
→
```

Can concurrent writes break the optimization?

current state:

*x	*y	n	m
2	42	1	42

optimized program:

```
  int n = *x;
  int m = *y;
→ int i = 0; int sum = m;
  while (i != n - 1) {
    i += 1; sum += m;
  }
  return sum;
```

||

```
  *x = 2;
  *y = 42;
→
```

Can concurrent writes break the optimization?

current state:

*x	*y	n	m	sum
2	42	1	42	42

optimized program:

```
  int n = *x;
  int m = *y;
  int i = 0; int sum = m;
  while (i != n - 1) {
    i += 1; sum += m;
  }
→ return sum;
```

||

```
  *x = 2;
  *y = 42;
→
```

The optimized program can produce the result 42
with initial *x = 1 and *y = 99
(by using the old value 1 of x and the new value 42 of y)

The unoptimized program cannot produce the result 42

current state:

*x	*y
1	99

unoptimized program:

```
int i = 0;
int sum = *y;
while (i != *x - 1) {
  i += 1; sum += *y;
}
return sum;
```

||

```
*x = 2;
*y = 42;
```

The unoptimized program cannot produce the result 42

current state:

*x	*y
1	99

unoptimized program:

```
int i = 0;
int sum = *y;
while (i != *x - 1) {
  i += 1; sum += *y;
}
return sum;
```

||

```
*x = 2;
*y = 42;
```

The unoptimized program cannot produce the result 42

current state:

*x	*y
2	42

unoptimized program:

```
int i = 0;
int sum = *y;
while (i != *x - 1) {
  i += 1; sum += *y;
}
return sum;
```

||

```
*x = 2;
*y = 42;
```

→

The unoptimized program cannot produce the result 42

current state:

*x	*y
2	42

unoptimized program:

```
int i = 0;
int sum = *y;
while (i != *x - 1) {
  i += 1; sum += *y;
}
return sum;
```

||

```
*x = 2;
*y = 42;
```

$\rightarrow$

The unoptimized program cannot produce the result 42

current state:

*x	*y
2	42

unoptimized program:

```
int i = 0;
int sum = *y;
while (i != *x - 1) {
  i += 1; sum += *y;
}
return sum;
```

||

```
*x = 2;
*y = 42;
```

→

The unoptimized program can **not** produce the result 42

with initial *x = 1 and *y = 99

(if the new value 42 of y is read, also the new value 2 of x is read)

The unoptimized program cannot produce the result 42

current state: *x *y

2 42

unoptimized program:

```
int i = 0;
}
return sum;
```

The optimization seems to introduce new program behavior!

The unoptimized program can **not** produce the result 42 with initial *x = 1 and *y = 99

(if the new value 42 of y is read, also the new value 2 of x is read)

Correctness under concurrency: relying on undefined behavior

unoptimized:

```
int i = 0; int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
}
return sum;
```

optimized:

```
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

||

```
*x = 2;
*y = 42;
```

Correctness under concurrency: relying on undefined behavior

unoptimized:

```
int i = 0; int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
}
return sum;
```

∥

```
*x = 2;
*y = 42;
```

optimized:

```
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

Data races are **undefined behavior (UB)** in C/C++/unsafe Rust.

Correctness under concurrency: relying on undefined behavior

unoptimized:

```
int i = 0; int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
}
return sum;
```

optimized:

```
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

||

~~*x = 2;~~
~~*y = 42;~~

Data races are **undefined behavior (UB)** in C/C++/unsafe Rust.

The compiler may assume their absence.

Correctness under concurrency: relying on undefined behavior

unoptimized:

```
int i = 0; int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
}
return sum;
```

The optimization is correct. But how can we prove that?

o

```
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

Data races are **undefined behavior (UB)** in C/C++/unsafe Rust.

The compiler may assume their absence.

The problem

How can we prove correctness of **concurrent** program optimizations relying on **data race UB** and involving **loops**?

	data race UB	**concurrent**	**loops**
[Ševčík, 2009], [Morisset et al., 2013]	✓	✓	∼
[Vafeiadis et al., 2015]	✓	✓	∼
CAS/Concurrent CompCert	∼	∼	✓
CompCertTSO [Ševčík et al., 2013]	✗	✓	✓
CCAL (CompCertX) [Gu et al., 2018]	✗	∼	✓
[Liang and Feng, 2016]	✗	✓	✓
ReLoC [Frumin et al., 2018]	✗	✓	✓
[Tassarotti et al., 2017]	✗	✓	✓
Transfinite Iris [Spies et al., 2021]	✗	✗	✓
Stacked Borrows [Jung et al., 2020]	✗	✗	∼

The problem

How can we prove correctness of **concurrent** program optimizations relying on **data race UB** and involving **loops**?

	data race UB	**concurrent**	**loops**
[Ševčík, 2009], [Morisset et al., 2013]	✓	✓	∼
[Vafeiadis et al., 2015]	✓	✓	∼
CAS/Concurrent CompCert	∼	∼	✓
CompCertTSO [Ševčík et al., 2013]	✗	✓	✓
CCAL (CompCertX) [Gu et al., 2018]	✗	∼	✓
[Liang and Feng, 2016]	✗	✓	✓
ReLoC [Frumin et al., 2018]	✗	✓	✓
[Tassarotti et al., 2017]	✗	✓	✓
Transfinite Iris [Spies et al., 2021]	✗	✗	✓
Stacked Borrows [Jung et al., 2020]	✗	✗	∼

The problem

How can we prove correctness of **concurrent** program optimizations relying on **data race UB** and involving **loops**?

	data race UB	**concurrent**	**loops**
[Ševčík, 2009], [Morisset et al., 2013]	✓	✓	~
[Vafeiadis et al., 2015]	✓	✓	~
CAS/Concurrent CompCert	~	~	✓
CompCertTSO [Ševčík et al., 2013]	✗	✓	✓
[Tassarotti et al., 2017]	✗	✓	✓
Transfinite Iris [Spies et al., 2021]	✗	✗	✓
Stacked Borrows [Jung et al., 2020]	✗	✗	~

- can only handle finite traces
- cannot handle potentially unbounded loops

The problem

How can we prove correctness of **concurrent** program optimizations relying on **data race UB** and involving **loops**?

	data race UB	concurrent	loops
[Ševčík, 2009], [Morisset et al., 2013]	✓	✓	∼
[Vafeiadis et al., 2015]	✓	✓	∼
CAS/Concurrent CompCert	∼	∼	✓
CompCertTSO [Ševčík et al., 2013]	✗	✓	✓
[Tassarotti et al., 2017]	✗	✓	✓
Transfinite Iris [Spies et al., 2021]	✗	✗	✓
Stacked Borrows [Jung et al., 2020]	✗	✗	∼

- no optimizations involving synchronizing operations (*e.g.*, atomic reads)

The problem

How can we prove correctness of **concurrent** program optimizations relying on **data race UB** and involving **loops**?

	data race UB	concurrent	loops
Our approach	✓	✓	✓
[Ševčík, 2009], [Morisset et al., 2013]	✓	✓	∼
[Vafeiadis et al., 2015]	✓	✓	∼
CAS/Concurrent CompCert	∼	∼	✓
CompCertTSO [Ševčík et al., 2013]	✗	✓	✓
CCAL (CompCertX) [Gu et al., 2018]	✗	∼	✓
[Liang and Feng, 2016]	✗	✓	✓
ReLoC [Frumin et al., 2018]	✗	✓	✓
[Tassarotti et al., 2017]	✗	✓	✓
Transfinite Iris [Spies et al., 2021]	✗	✗	✓
Stacked Borrows [Jung et al., 2020]	✗	✗	∼

Key idea: ownership acquisition on unsynchronized accesses

unoptimized:

optimized:

$$\{\text{True}\}$$

```
int i = 0; int sum = *y;
while (i != *x - 1) {
  i += 1; sum += *y;
}
return sum;
```

$\succeq$

```
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

Key idea: ownership acquisition on unsynchronized accesses

unoptimized:

optimized:

reach unsynchronized access to y

$\{\text{True}\}$

```
int i = 0; int sum = *y;
while (i != *x - 1) {
  i += 1; sum += *y;
}
return sum;
```

$\succeq$

```
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

Key idea: ownership acquisition on unsynchronized accesses

unoptimized:

reach unsynchronized access to y

```
int i = 0; int sum = *y;
while (i != *x - 1) {
  i += 1; sum += *y;
}
return sum;
```

$\succeq$

$\{\mathsf{True}\}$

$\{y \mapsto_{\mathsf{src}} z_y * y \mapsto_{\mathsf{tgt}} z_y\}$

optimized:

obtain ownership with proof rule

```
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

Key idea: ownership acquisition on unsynchronized accesses

unoptimized:

optimized:

reach unsynchronized access to x

obtain ownership with proof rule

$$\{\text{True}\}$$

$$\{y \mapsto^{\text{src}} z_y * y \mapsto^{\text{tgt}} z_y\}$$

```
int i = 0; int sum = *y;
while (i != *x - 1) {
  i += 1; sum += *y;
}
return sum;
```

$\succeq$

```
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

Key idea: ownership acquisition on unsynchronized accesses

unoptimized:

optimized:

reach unsynchronized access to x

obtain ownership with proof rule

$$\{\mathsf{True}\}$$

$$\{y \mapsto^{\mathsf{src}} z_y * y \mapsto^{\mathsf{tgt}} z_y * x \mapsto^{\mathsf{src}} z_x * x \mapsto^{\mathsf{tgt}} z_x\}$$

```
int i = 0; int sum = *y;
while (i != *x - 1) {
  i += 1; sum += *y;
}
return sum;
```

$\succeq$

```
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

Key idea: ownership acquisition on unsynchronized accesses

unoptimized:

optimized:

obtain ownership
with proof rule

$$\{\mathrm{True}\}$$

$$\{y \mapsto^{\mathsf{src}} z_y * y \mapsto^{\mathsf{tgt}} z_y * x \mapsto^{\mathsf{src}} z_x * x \mapsto^{\mathsf{tgt}} z_x\}$$

```
int i = 0; int sum = *y;
while (i != *x - 1) {
  i += 1; sum += *y;
}
return sum;
```

$\succeq$

```
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

$$\{y \mapsto^{\mathsf{src}} z_y * y \mapsto^{\mathsf{tgt}} z_y * x \mapsto^{\mathsf{src}} z_x * x \mapsto^{\mathsf{tgt}} z_x\}$$

retain ownership
throughout the loop

Simuliris: a separation logic-based simulation framework

Simuliris: separation logic-based simulation framework

- soundness: fair termination-preserving contextual refinement
- proof rules for verifying optimizations: coinduction, . . .

Simuliris: a separation logic-based simulation framework

logic for data race
based optimizations

Stacked Borrows for Rust
[Jung et al., 2020] + concurrency

Simuliris: separation logic-based simulation framework

- soundness: fair termination-preserving contextual refinement
- proof rules for verifying optimizations: coinduction, ...

Simuliris: a separation logic-based simulation framework

logic for data race
based optimizations

Stacked Borrows for Rust
[Jung et al., 2020] + concurrency

Simuliris: separation logic-based simulation framework

- soundness: fair termination-preserving contextual refinement
- proof rules for verifying optimizations: coinduction, ...

fully mechanized in the Coq proof assistant

based on the Iris framework

Simuliris: a separation logic-based simulation framework

logic for data race
based optimizations

Stacked Borrows for Rust
[Jung et al., 2020] + concurrency

Simuliris: separation logic-based simulation framework

- soundness: fair termination-preserving contextual refinement
- proof rules for verifying optimizations: coinduction, . . .

fully mechanized in the Coq proof assistant

based on the Iris framework

Simuliris: a separation logic-based simulation framework

logic for data race
based optimizations

Stacked Borrows for Rust
[Jung et al., 2020] + concurrency

Simuliris: separation logic-based simulation framework

- soundness: fair termination-preserving contextual refinement
- proof rules for verifying optimizations: coinduction, . . .

fully mechanized in the Coq proof assistant

based on the Iris framework

Key ingredient: a powerful simulation relation

traditional coinductive simulation + modern separation logic

Key ingredient: a powerful simulation relation

$$\{P\}\ e_s \succeq e_t\ \{v_s, v_t.\ Q\}$$

source expression (unoptimized)

target expression (optimized)

coinductive simulation

separation logic

Key ingredient: a powerful simulation relation

$$\{P\}\ e_s \succeq e_t\ \{v_s, v_t.\ Q\}$$

source expression (unoptimized) → e_s

target expression (optimized) → e_t

coinductive simulation

- coinduction
- reasoning about UB
- flexible stuttering

separation logic

Key ingredient: a powerful simulation relation

precondition

postcondition

$$\{P\}\ e_s \succeq e_t\ \{v_s, v_t.\ Q\}$$

source expression (unoptimized)

target expression (optimized)

coinductive simulation

- coinduction
- reasoning about UB
- flexible stuttering

separation logic

- compositional proof rules

Key ingredient: a powerful simulation relation

precondition

postcondition

$$\{P\}\ e_s \succeq e_t\ \{v_s, v_t.\ Q\}$$

source expression (unoptimized)

target expression (optimized)

coinductive simulation

- coinduction
- reasoning about UB
- flexible stuttering

separation logic

- compositional proof rules
- ownership reasoning with custom resources Iris

Key ingredient: a powerful simulation relation

$$\{P\}\ e_s \succeq e_t\ \{v_s, v_t.\ Q\}$$

coinductive simulation

- coinduction
- reasoning about UB
- flexible stuttering

separation logic

- compositional proof rules
- ownership reasoning with custom resources

Source and target reasoning with flexible stuttering

$$\{P\}\ e_s \succeq e_t\ \{v_s, v_t.\ Q\}$$

Source and target reasoning with flexible stuttering

$$\{P\}\ e_s \succeq e_t\ \{v_s, v_t.\ Q\}$$

focus source

focus target

$$\{P\}\ e_s\ \{e'_s.\ R\}^{\text{src}} \qquad \{P\}\ e_t\ \{e'_t.\ R\}^{\text{tgt}}$$

with rules of total separation logic

Source and target reasoning with flexible stuttering

$$\{P\}\ e_s \succeq e_t\ \{v_s, v_t.\ Q\}$$

focus source

focus target

$$\{P\}\ e_s\ \{e_s'.\ R\}^{\text{src}} \qquad \{P\}\ e_t\ \{e_t'.\ R\}^{\text{tgt}}$$

with rules of total separation logic

Enabled by a flexible implicit stuttering mechanism without explicit step counting!

Key ingredient: a powerful simulation relation

$$\{P\}\ e_s \succeq e_t\ \{v_s, v_t.\ Q\}$$

coinductive simulation

- coinduction
- reasoning about UB
- flexible stuttering

separation logic

- compositional proof rules
- ownership reasoning
  with custom resources Iris

Ownership is useful for justifying optimizations

$$
\begin{array}{lcl}
\textsf{let } x := \textsf{new}(42) \textsf{ in} & & \textsf{let } x := \textsf{new}(42) \textsf{ in} \\
\textsf{call } f\ (); & \succeq & \textsf{call } f\ (); \\
{*}x & & 42
\end{array}
$$

Ownership is useful for justifying optimizations

$\{\text{True}\}$

let x := new(42) in		let x := new(42) in
call f ();	$\succeq$	call f ();
*x		42

$\{v_s, v_t.\, v_s = v_t = 42\}$

Ownership is useful for justifying optimizations

$$\{\mathsf{True}\}$$

$$\begin{array}{lcl} \mathsf{let}\ x := \mathsf{new}(42)\ \mathsf{in} & & \mathsf{let}\ x := \mathsf{new}(42)\ \mathsf{in} \\ & \{x \mapsto^{\mathsf{src}} 42 * x \mapsto^{\mathsf{tgt}} 42\} & \\ \mathsf{call}\ f\ (); & \succeq & \mathsf{call}\ f\ (); \\ & & \\ {*}x & & 42 \end{array}$$

$$\{v_s, v_t.\, v_s = v_t = 42\}$$

Ownership is useful for justifying optimizations

$$\{\mathsf{True}\}$$

let x := new(42) in		let x := new(42) in
	$\{x \mapsto^{src} 42 * x \mapsto^{tgt} 42\}$	
call f ();	$\succeq$	call f ();
	$\{x \mapsto^{src} 42 * x \mapsto^{tgt} 42\}$	
*x		42
	$\{v_s, v_t.\, v_s = v_t = 42\}$	

Ownership is useful for justifying optimizations

$$\{\mathsf{True}\}$$

$$\begin{array}{lcl}
\mathsf{let}\ x := \mathsf{new}(42)\ \mathsf{in} & & \mathsf{let}\ x := \mathsf{new}(42)\ \mathsf{in} \\
& \{x \mapsto^{\mathsf{src}} 42 * x \mapsto^{\mathsf{tgt}} 42\} & \\
\mathsf{call}\ f\ (); & \succeq & \mathsf{call}\ f\ (); \\
& \{x \mapsto^{\mathsf{src}} 42 * x \mapsto^{\mathsf{tgt}} 42\} & \\
{*}x & & 42
\end{array}$$

$$\{v_s, v_t.\ v_s = v_t = 42\}$$

Unknown code must respect the ownership principles of our logic!

Key ingredient: a powerful simulation relation

$$\{P\}\ e_s \succeq e_t\ \{v_s, v_t.\ Q\}$$

coinductive simulation

- coinduction
- reasoning about UB
- flexible stuttering

separation logic

- compositional proof rules
- ownership reasoning
  with custom resources

Interaction with coinduction

$$\{\mathsf{True}\}$$

```
let x := new(42) in                  let x := new(42) in

while call f (*x) do        ⪰        while call f (42) do
  ()                                   ()
od                                   od
```

$$\{v_s, v_t.\, v_s = v_t = ()\}$$

Interaction with coinduction

$$\{\mathsf{True}\}$$

let x := new(42) in		let x := new(42) in
	$\{x \mapsto^{\mathsf{src}} 42 * x \mapsto^{\mathsf{tgt}} 42\}$	
while call f (*x) do	$\succeq$	while call f (42) do
()		()
od		od

$$\{v_s, v_t.\, v_s = v_t = ()\}$$

Reasoning about loops: coinduction

$$W_s = \textsf{while } c_s \textsf{ do } e_s \textsf{ od} \qquad W_t = \textsf{while } c_t \textsf{ do } e_t \textsf{ od}$$

$$\frac{}{\{I\}\ W_s \succeq W_t\ \{v_s, v_t.\ Q\}}$$

loop invariant

Reasoning about loops: coinduction

$$W_s = \textsf{while}\ c_s\ \textsf{do}\ e_s\ \textsf{od} \qquad W_t = \textsf{while}\ c_t\ \textsf{do}\ e_t\ \textsf{od}$$

new proof goal

$$\frac{\{I\}\ \textsf{if}\ c_s\ \textsf{then}\ e_s; W_s\ \textsf{else}\ () \succeq \textsf{if}\ c_t\ \textsf{then}\ e_t; W_t\ \textsf{else}\ ()\ \{e_s', e_t'.\ (\exists v_s, v_t.\ e_s' = v_s * e_t' = v_t * Q)\}}{\{I\}\ W_s \succeq W_t\ \{v_s, v_t.\ Q\}}$$

loop invariant

Reasoning about loops: coinduction

$$W_s = \text{while } c_s \text{ do } e_s \text{ od} \qquad W_t = \text{while } c_t \text{ do } e_t \text{ od}$$

new proof goal

allows use of coinduction hypothesis

$$\frac{\{I\}\ \text{if } c_s \text{ then } e_s; W_s \text{ else } () \succeq \text{if } c_t \text{ then } e_t; W_t \text{ else } ()\ \{e'_s, e'_t.\ (\exists v_s, v_t.\ e'_s = v_s * e'_t = v_t * Q) \lor (e'_s = W_s * e'_t = W_t * I)\}}{\{I\}\ W_s \succeq W_t\ \{v_s, v_t.\ Q\}}$$

loop invariant

Interaction with coinduction

$$\{\text{True}\}$$

let x := new(42) in		let x := new(42) in
	$\{x \mapsto_{\text{src}} 42 * x \mapsto_{\text{tgt}} 42\}$	
while call f (*x) do	$\succeq$	while call f (42) do
()		()
od		od

$$\{v_s, v_t.\ v_s = v_t = ()\}$$

Pick invariant $I \triangleq x \mapsto_{\text{src}} 42 * x \mapsto_{\text{tgt}} 42$

Interaction with coinduction

$\{\mathsf{True}\}$

let x := new(42) in let x := new(42) in

$\{x \mapsto_{src} 42 * x \mapsto_{tgt} 42\}$

Ownership reasoning is a powerful tool in combination with coinductive simulations!

od od

$\{v_s, v_t.\, v_s = v_t = ()\}$

Pick invariant $I \triangleq x \mapsto_{src} 42 * x \mapsto_{tgt} 42$

Key ingredient: a powerful simulation relation

$$\{P\}\ e_s \succeq e_t\ \{v_s, v_t.\ Q\}$$

coinductive simulation

- coinduction
- reasoning about UB
- flexible stuttering

separation logic

- compositional proof rules
- ownership reasoning
  with custom resources

Simuliris: a separation logic-based simulation framework

logic for data race based optimizations

Stacked Borrows for Rust [Jung et al., 2020] + concurrency

Simuliris: separation logic-based simulation framework

- soundness: fair termination-preserving contextual refinement
- proof rules for verifying optimizations: coinduction, . . .

fully mechanized in the Coq proof assistant

based on the Iris framework

Optimizations with external locations: motivating example


```
fn foo(x) {                    fn foo_opt(x) {
  x ← 41;
  x ← 42;            ⪰           x ← 42;
  *x                             42
}                              }
```

Optimizations with external locations: motivating example

in separation logic verification: assume ownership in precondition

$$\{x \mapsto^{\text{src}} z * x \mapsto^{\text{tgt}} z\}$$

```
fn foo(x) {
  x ← 41;
  x ← 42;
  *x
}
```

$\succeq$

```
fn foo_opt(x) {

  x ← 42;
  42
}
```

Optimizations with external locations: motivating example

in separation logic verification: assume ownership in precondition

fn foo(x) { $\qquad \{x \mapsto^{src} z * x \mapsto^{tgt} z\}$ $\qquad$ fn foo_opt(x) {

$x \leftarrow 41;$

$x \leftarrow 42;$ $\qquad \succeq \qquad$ $x \leftarrow 42;$

$*x$ $\qquad$ 42

} $\qquad$ }

in compiler optimizations: surrounding code is **not cooperative**!

fn foo(x) { $\qquad \{??\}$ $\qquad$ fn foo_opt(x) {

$x \leftarrow 41;$

$x \leftarrow 42;$ $\qquad \succeq \qquad$ $x \leftarrow 42;$

$*x$ $\qquad$ 42

} $\qquad$ }

Optimizations with external locations: motivating example

in separation logic verification: assume ownership in precondition

$$\{x \mapsto^{\text{src}} z * x \mapsto^{\text{tgt}} z\}$$

```
fn foo(x) {              fn foo_opt(x) {
  x ← 41;
  x ← 42;        ⪰          x ← 42;
  *x                        42
}                         }
```

in compiler optimizations: surrounding code is **not cooperative**!

$$\{x_s \approx x_t\}$$

```
fn foo(x) {              fn foo_opt(x) {
  x ← 41;
  x ← 42;        ⪰          x ← 42;
  *x                        42
}                         }
```

Interaction protocol with unknown code: public value relation

contract: similar values $v_s \approx v_t$ in source and target

for integers: $z_s \approx z_t \triangleq z_s = z_t$

for memory locations $\ell_s \approx \ell_t$:

- **contract**: stored values are related by $\approx$
- **accessible by anyone** as long as the contract is observed

Interaction protocol with unknown code: public value relation

contract: similar values $v_s \approx v_t$ in source and target

for integers: $z_s \approx z_t \triangleq z_s = z_t$

for memory locations $\ell_s \approx \ell_t$:

- **contract**: stored values are related by $\approx$
- **accessible by anyone** as long as the contract is observed

How can we use this to justify optimizations?

Interaction protocol with unknown code: public value relation

contract: similar values $v_s \approx v_t$ in source and target

Idea: we can break the contract as long as no other thread will notice

- **contract**: stored values are related by $\approx$
- **accessible by anyone** as long as the contract is observed

How can we use this to justify optimizations?

Ownership acquisition on unsynchronized accesses

When the source program does an unsynchronized access to $\ell_s \approx \ell_t$, we temporarily obtain ownership of ℓ_s and ℓ_t.

Ownership acquisition on unsynchronized accesses

When the source program does an unsynchronized access to $\ell_s \approx \ell_t$, we temporarily obtain ownership of ℓ_s and ℓ_t.

An **unsynchronized write** $\ell_s \leftarrow \_$ is reachable:
$\Rightarrow$ *all* concurrent accesses would be conflicting
$\Rightarrow$ obtain **exclusive** ownership $\ell_s \mapsto^{\text{src}} v_s, \ell_t \mapsto^{\text{tgt}} v_t$

Acquiring ownership on writes, formally: first attempt

$$\frac{}{\{\ell_s \approx \ell_t * P\}\; K[\,\ell_s \leftarrow v_0\,] \succeq e_t \;\{\Phi\}}$$

public locations

unsynchronized write in the source

Acquiring ownership on writes, formally: first attempt

obtain ownership

$$\frac{\forall v_s, v_t.\ \{\ell_s \mapsto^{\text{src}} v_s * \ell_t \mapsto^{\text{tgt}} v_t * v_s \approx v_t * P\}\ K[\,\ell_s \leftarrow v_0\,] \succeq e_t\ \{\Phi\}}{\{\ell_s \approx \ell_t * P\}\ K[\,\ell_s \leftarrow v_0\,] \succeq e_t\ \{\Phi\}}$$

public locations

unsynchronized write in the source

Proving the optimization with ownership

$\{x_s \approx x_t\}$

$x_s \leftarrow 41;$

$x_s \leftarrow 42; \quad \succeq \quad x_t \leftarrow 42;$

$*x_s \quad\quad 42$

Proving the optimization with ownership

$$\{x_s \approx x_t\}$$

$$\{x_s \mapsto^{\text{src}} z * x_t \mapsto^{\text{tgt}} z\}$$

$x_s \leftarrow 41;$		
$x_s \leftarrow 42;$	$\succeq$	$x_t \leftarrow 42;$
$*x_s$		42

Proving the optimization with ownership

$$\{x_s \approx x_t\}$$

$$\{x_s \mapsto^{\text{src}} z * x_t \mapsto^{\text{tgt}} z\}$$

$x_s \leftarrow 41;$

contract temporarily broken

$$\{x_s \mapsto^{\text{src}} 41 * x_t \mapsto^{\text{tgt}} z\}$$

$x_s \leftarrow 42;$	$\succeq$	$x_t \leftarrow 42;$
$*x_s$		42

Proving the optimization with ownership

$$\{x_s \approx x_t\}$$

$$\{x_s \mapsto^{\text{src}} z * x_t \mapsto^{\text{tgt}} z\}$$

$x_s \leftarrow 41;$

$$\{x_s \mapsto^{\text{src}} 41 * x_t \mapsto^{\text{tgt}} z\}$$

$x_s \leftarrow 42;$ $\quad \succeq \quad$ $x_t \leftarrow 42;$

$$\{x_s \mapsto^{\text{src}} 42 * x_t \mapsto^{\text{tgt}} 42\}$$

$*x_s$ $\quad\quad$ 42

Proving the optimization with ownership

$$\{x_s \approx x_t\}$$

$$\{x_s \mapsto^{\text{src}} z * x_t \mapsto^{\text{tgt}} z\}$$

$x_s \leftarrow 41;$

$$\{x_s \mapsto^{\text{src}} 41 * x_t \mapsto^{\text{tgt}} z\}$$

$x_s \leftarrow 42;$ $\quad\succeq\quad$ $x_t \leftarrow 42;$

$$\{x_s \mapsto^{\text{src}} 42 * x_t \mapsto^{\text{tgt}} 42\}$$

$*x_s$ $\qquad$ 42

$$\{v_s, v_t.\, v_s = v_t = 42 * x_s \approx x_t\}$$

Proving the optimization with ownership

$$\{x_s \approx x_t\}$$

$$\{x_s \mapsto^{\text{src}} z * x_t \mapsto^{\text{tgt}} z\}$$

$$\{x_s \mapsto^{\text{src}} z' * x_t \mapsto^{\text{tgt}} z' * x_s \mapsto^{\text{src}} z * x_t \mapsto^{\text{tgt}} z\}$$

$x_s \leftarrow 41;$

$$\{x_s \mapsto^{\text{src}} 41 * x_t \mapsto^{\text{tgt}} z\}$$

$x_s \leftarrow 42;$ $\quad\succeq\quad$ $x_t \leftarrow 42;$

$$\{x_s \mapsto^{\text{src}} 42 * x_t \mapsto^{\text{tgt}} 42\}$$

$*x_s$ $\qquad$ 42

$$\{v_s, v_t.\, v_s = v_t = 42 * x_s \approx x_t\}$$

What prevents us from acquiring ownership multiple times?

Proving the optimization with ownership

$$\{x_s \approx x_t\}$$

$$\{x_s \mapsto^{\mathsf{src}} z * x_t \mapsto^{\mathsf{tgt}} z\}$$

$$\{x_s \mapsto^{\mathsf{src}} z' * x_t \mapsto^{\mathsf{tgt}} z' * x_s \mapsto^{\mathsf{src}} z * x_t \mapsto^{\mathsf{tgt}} z\}$$

$x_s \leftarrow 41;$

$$\{x_s \mapsto^{\mathsf{src}} 41 * x_t \mapsto^{\mathsf{tgt}} z\}$$

The acquisition rule is unsound! ☹

$$\{x_s \mapsto^{\mathsf{src}} 42 * x_t \mapsto^{\mathsf{tgt}} 42\}$$

$*x_s$ 42

$$\{v_s, v_t.\, v_s = v_t = 42 * x_s \approx x_t\}$$

What prevents us from acquiring ownership multiple times?

Solution: avoiding duplication of ownership

Track locations exploited by the current thread π: $\text{exploit}_\pi\ C$

obtain ownership

$$\frac{\forall v_t, v_s.\ \{\ell_s \mapsto^{\text{src}} v_s * \ell_t \mapsto^{\text{tgt}} v_t * v_s \approx v_t * \qquad P\}\ K[\,\ell_s \leftarrow v_0\,] \succeq_\pi e_t\ \{\Phi\}}{\{\ell_s \approx \ell_t * \qquad P\}\ K[\,\ell_s \leftarrow v_0\,] \succeq_\pi e_t\ \{\Phi\}}$$

public locations

unsynchronized write
in the source

Solution: avoiding duplication of ownership

Track locations exploited by the current thread π: $\text{exploit}_\pi\ C$

exploit once

obtain ownership

$\ell_s \notin C$

remember ℓ_s

$$\frac{\forall v_t, v_s.\ \{\ell_s \mapsto^{\text{src}} v_s * \ell_t \mapsto^{\text{tgt}} v_t * v_s \approx v_t * \text{exploit}_\pi\ (C, \ell_s \mapsto W) * P\}\ K[\ell_s \leftarrow v_0] \succeq_\pi e_t\ \{\Phi\}}{\{\ell_s \approx \ell_t * \text{exploit}_\pi\ C * P\}\ K[\ell_s \leftarrow v_0] \succeq_\pi e_t\ \{\Phi\}}$$

public locations

track exploited locations
for current thread

unsynchronized write
in the source

Solution: avoiding duplication of ownership

Track locations exploited by the current thread π: $\text{exploit}_\pi\ C$

defined with custom Iris ghost state

exploit once

obtain ownership

remember ℓ_s

$$\frac{\ell_s \notin C \qquad \forall v_t, v_s.\ \{\ell_s \mapsto^{\text{src}} v_s * \ell_t \mapsto^{\text{tgt}} v_t * v_s \approx v_t * \text{exploit}_\pi\ (C, \ell_s \mapsto W) * P\}\ K[\,\ell_s \leftarrow v_0\,] \succeq_\pi e_t\ \{\Phi\}}{\{\ell_s \approx \ell_t * \text{exploit}_\pi\ C * P\}\ K[\,\ell_s \leftarrow v_0\,] \succeq_\pi e_t\ \{\Phi\}}$$

public locations

track exploited locations for current thread

unsynchronized write in the source

Solution: avoiding duplication of ownership

Track locations exploited by the current thread π: $\text{exploit}_\pi\ C$

defined with custom Iris ghost state

generalization: unsynchronized access just needs to be reachable (not the directly next instruction)

obtain ownership

$\ell_s \notin C$

remember ℓ_s

$$\frac{\forall v_t, v_s.\ \{\ell_s \mapsto^{\text{src}} v_s * \ell_t \mapsto^{\text{tgt}} v_t * v_s \approx v_t * \text{exploit}_\pi\ (C, \ell_s \mapsto W) * P\}\ K[\ell_s \leftarrow v_0] \succeq_\pi e_t\ \{\Phi\}}{\{\ell_s \approx \ell_t * \text{exploit}_\pi\ C * P\}\ K[\ell_s \leftarrow v_0] \succeq_\pi e_t\ \{\Phi\}}$$

public locations

track exploited locations for current thread

unsynchronized write in the source

Solution: avoiding duplication of ownership

Track locations exploited by the current thread π: $\text{exploit}_\pi\ C$

defined with custom Iris ghost state

generalization: unsynchronized access just needs to be reachable (not the directly next instruction)

obtain ownership

remember ℓ_s

rule for reads: obtain fractional (read-only) ownership

public locations

track exploited locations for current thread

unsynchronized write in the source

Maintaining & releasing ownership on synchronization

We can maintain ownership until the thread **observably** synchronizes.

action (potentially) visible by other threads

Maintaining & releasing ownership on synchronization

We can maintain ownership until the thread **observably** synchronizes.

action (potentially) visible by other threads

Example: rule for atomic writes

$$\{\ell_s \approx \ell_t * v_s \approx v_t * \text{exploit}_\pi\ \emptyset\}$$
$$\ell_s \leftarrow^{sc} v_s \succeq_\pi \ell_t \leftarrow^{sc} v_t$$
$$\{v_s', v_t'.\ \text{exploit}_\pi\ \emptyset\}$$

Verifying the motivating example


```
int i = 0; int sum = *y;
while (i != *x) {
    i += 1; sum += *y;
}
return sum;
```

```
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n) {
    i += 1; sum += m;
}
return sum;
```

Verifying the motivating example

$$\{x_s \approx x_t * y_s \approx y_t\}$$

		let (m, n) := (*y, *x) in
let (i, sum) := (new(0), new(*y)) in		let (i, sum) := (new(0), new(m)) in
while *i ≠ *x do	$\succeq_\pi$	while *i ≠ n do
$i \leftarrow$ *i + 1;		$i \leftarrow$ *i + 1;
sum ← *sum + *y		sum ← *sum + m
od; *sum		od; *sum

$$\{v_s, v_t.\, v_s \approx v_t\}$$

1. Obtain ownership of x and y due to unsynchronized reads in the source
2. Initiate coinduction

Simuliris: a separation logic-based simulation framework

logic for data race based optimizations

Stacked Borrows for Rust [Jung et al., 2020] + concurrency

Simuliris: separation logic-based simulation framework

- soundness: fair termination-preserving contextual refinement
- proof rules for verifying optimizations: coinduction, ...

fully mechanized in the Coq proof assistant

based on the Iris framework

Soundness: fair termination-preserving contextual refinement

fair termination-preserving contextual refinement

Soundness: fair termination-preserving contextual refinement

fair termination-preserving contextual refinement

optimized expression		unoptimized expression
can terminate with result	$\Longrightarrow$	can terminate with similar result

Soundness: fair termination-preserving contextual refinement

fair termination-preserving contextual refinement

optimized expression		unoptimized expression
can terminate with result	$\Longrightarrow$	can terminate with similar result
can diverge *under fair scheduling*	$\Longrightarrow$	can diverge *under fair scheduling*

Soundness: fair termination-preserving contextual refinement

fair termination-preserving contextual refinement

for any surrounding program, assuming no UB in unoptimized program

optimized expression		unoptimized expression
can terminate with result	$\Longrightarrow$	can terminate with similar result
can diverge *under fair scheduling*	$\Longrightarrow$	can diverge *under fair scheduling*

Soundness: fair termination-preserving contextual refinement

fair termination-preserving contextual refinement

for any surrounding program, assuming no UB in unoptimized program

optimized expression		unoptimized expression
can terminate with result	$\Longrightarrow$	can terminate with similar result
can diverge *under fair scheduling*	$\Longrightarrow$	can diverge *under fair scheduling*

Core soundness proof: proved once and for all!

More in the paper . . .

logic for data race
based optimizations

Stacked Borrows for Rust
[Jung et al., 2020] + concurrency

Simuliris: separation logic-based simulation framework

- soundness: fair termination-preserving contextual refinement
- proof rules for verifying optimizations: coinduction, . . .

fully mechanized in the Coq proof assistant

based on the Iris framework

Simuliris: a separation logic-based simulation framework

logic for data race
based optimizations

Stacked Borrows for Rust
[Jung et al., 2020] + concurrency

Simuliris: separation logic-based simulation framework

- soundness: fair termination-preserving contextual refinement
- proof rules for verifying optimizations: coinduction, ...

fully mechanized in the Coq proof assistant

based on the Iris framework

https://gitlab.mpi-sws.org/iris/simuliris

Thanks for listening!

Generalization: reachability of an unsynchronized write

The argument even works if the write is just
(unconditionally) reachable!

exploit once

reach unsynchronized write in the source

obtain ownership

$$\frac{\ell_s \notin C \qquad e_s \rightarrow^*_? K[\,\ell_s \leftarrow v_0\,] \qquad \forall v_t, v_s.\ \{\ell_s \mapsto^{\text{src}} v_s * \ell_t \mapsto^{\text{tgt}} v_t * v_s \approx v_t * \text{exploit}_\pi\ (C, \ell_s \mapsto W) * P\}\ e_s \succeq_\pi e_t\ \{\Phi\}}{\{\ell_s \approx \ell_t * \text{exploit}_\pi\ C * P\}\ e_s \succeq_\pi e_t\ \{\Phi\}}$$

public locations

track exploited locations for current thread

Generalization: reachability of an unsynchronized write

The argument even works if the write is just
(unconditionally) reachable!

exploit once → $\ell_s \notin C$

reach unsynchronized write in the source → $e_s \rightarrow^*_? K[\,\ell_s \leftarrow v_0\,]$

obtain ownership → $\ell_s \mapsto^{\mathsf{src}} v_s * \ell_t \mapsto^{\mathsf{tgt}} v_t$

$$\frac{\ell_s \notin C \qquad e_s \rightarrow^*_? K[\,\ell_s \leftarrow v_0\,] \qquad \forall v_t, v_s.\ \{\ell_s \mapsto^{\mathsf{src}} v_s * \ell_t \mapsto^{\mathsf{tgt}} v_t * v_s \approx v_t * \mathsf{exploit}_\pi\ (C, \ell_s \mapsto W) * P\}\ e_s \succeq_\pi e_t\ \{\Phi\}}{\{\ell_s \approx \ell_t * \mathsf{exploit}_\pi\ C * P\}\ e_s \succeq_\pi e_t\ \{\Phi\}}$$

public locations → $\ell_s \approx \ell_t$

track exploited locations for current thread → $\mathsf{exploit}_\pi\ C$

A similar rule holds for reads!

Ownership acquisition on unsynchronized accesses

When the source program does an unsynchronized access to $\ell_s \approx \ell_t$, we temporarily obtain ownership of ℓ_s and ℓ_t.

An unsynchronized read $*\ell_s$ is reachable:
$\Rightarrow$ concurrent *write* accesses would be conflicting
$\Rightarrow$ obtain **fractional** ownership $\ell_s \mapsto^{\text{src}}_{q} v_s, \ell_t \mapsto^{\text{tgt}}_{q} v_t$

Ownership acquisition on unsynchronized accesses

When the source program does an unsynchronized access to $\ell_s \approx \ell_t$, we temporarily obtain ownership of ℓ_s and ℓ_t.

Do we ever have to give up ownership again?

$\Rightarrow$ concurrent *write* accesses would be conflicting

$\Rightarrow$ obtain **fractional** ownership $\ell_s \mapsto^{src}_{q} v_s, \ell_t \mapsto^{tgt}_{q} v_t$

Rule for unsynchronized reads

reach unsynchronized read in the source

exploit once

obtain ownership

$$\frac{\ell_s \notin C \qquad e_s \rightarrow^*_? K[\,*\ell_s\,] \qquad \forall v_t, v_s, q.\ \{\ell_s \mapsto^{\text{src}}_q v_s * \ell_t \mapsto^{\text{tgt}}_q v_t * v_s \approx v_t * \text{exploit}_\pi\ (C, \ell_s \mapsto R(q)) * P\}\ e_s \succeq_\pi e_t\ \{\Phi\}}{\{\ell_s \approx \ell_t * \text{exploit}_\pi\ C * P\}\ e_s \succeq_\pi e_t\ \{\Phi\}}$$

public locations

track exploited locations for current thread

Releasing ownership

$$\frac{C(\ell_s) = W \qquad \{\mathsf{exploit}_\pi\ (C \setminus \ell_s) * P\}\ e_s \succeq_\pi e_t\ \{\Phi\}}{\{\ell_s \mapsto^{\mathsf{src}} v_s * \ell_t \mapsto^{\mathsf{tgt}} v_t * v_s \approx v_t * \ell_s \approx \ell_t * \mathsf{exploit}_\pi\ C * P\}\ e_s \succeq_\pi e_t\ \{\Phi\}}$$

Fair termination-preserving refinement

The compiler should not be allowed to perform the following transformation:

```
while !lock(l) do        ||   if lock(l)          ⪰   while true do   ||   if lock(l)
   ()                    ||   then unlock(l)            ()            ||   then unlock(l)
od;                      ||   else ()                 od              ||   else ()
unlock(l)
```

The source program only has diverging executions under an unfair scheduler, while the target program diverges even under a fair scheduler.

In the paper: proofs of optimizations relying on data races

- optimization hoisting reads out of a while loop

$$\begin{array}{l}\mathsf{let}\ (\mathrm{i}, \mathrm{sum}) := (\mathsf{new}(0), \mathsf{new}(*y_s))\ \mathsf{in}\\ \mathsf{while}\ *\mathrm{i} \neq *x_s\ \mathsf{do}\\ \quad i \leftarrow *\mathrm{i}+1; \mathrm{sum} \leftarrow *\mathrm{sum} + *y_s\\ \mathsf{od};\ *\mathrm{sum}\end{array} \quad \succeq_\pi \quad \begin{array}{l}\mathsf{let}\ (\mathrm{n}, \mathrm{m}) := (*x_t, *y_t)\ \mathsf{in}\\ \mathsf{let}\ (\mathrm{i}, \mathrm{sum}) := (\mathsf{new}(0), \mathsf{new}(\mathrm{m}))\ \mathsf{in}\\ \mathsf{while}\ *\mathrm{i} \neq \mathrm{n}\ \mathsf{do}\\ \quad i \leftarrow *\mathrm{i}+1; \mathrm{sum} \leftarrow *\mathrm{sum} + \mathrm{m}\\ \mathsf{od};\ *\mathrm{sum}\end{array}$$

Requires reasoning about potentially infinite loops!

In the paper: proofs of optimizations relying on data races

- optimization hoisting reads out of a while loop
- optimization eliminating reads and writes over unknown read-only code

$$
\begin{array}{lcl}
x_s \leftarrow 42; & & x_t \leftarrow 42; \\
e^{\mathrm{RO}}; & \succeq_\pi & e^{\mathrm{RO}}; \\
{*}x_s & & 42
\end{array}
$$

In the paper: proofs of optimizations relying on data races

- optimization hoisting reads out of a while loop
- optimization eliminating reads and writes over unknown read-only code

$$
\begin{array}{lcl}
x_s \leftarrow 42; & & x_t \leftarrow 42; \\
*^{sc} y_s; & \succeq_\pi & *^{sc} y_t; \\
*x_s & & 42
\end{array}
$$

Not supported by CAS/Concurrent CompCert!

In the paper: proofs of optimizations relying on data races

- optimization hoisting reads out of a while loop
- optimization eliminating reads and writes over unknown read-only code
- eliminations and reorderings using data races by [Ševčík, 2009]

In the paper: new optimization proofs for Stacked Borrows

Stacked Borrows: an experimental aliasing model for Rust

- determines which kinds of memory accesses are allowed
- enables powerful optimizations

In the paper: new optimization proofs for Stacked Borrows

Stacked Borrows: an experimental aliasing model for Rust

- determines which kinds of memory accesses are allowed
- enables powerful optimizations

Using Simuliris, we have...

- extended the optimization proofs by [Jung et al., 2020] to concurrent environments
- developed a new proof of an optimization involving loops:

```
// x: &i32, g: &Fn() -> (),
// f: &Fn(i32) -> bool
while f(*x) {
  g();
}
```

$\succeq$

```
let r = *x;
while f(r) {
  g();
}
```