Relational Thread-Modular Abstract Interpretation Under Relaxed Memory Models

Thibault Suzanne [1]

Antoine Miné [2]

3rd December 2018

[1]École Normale Supérieure, PSL Research University, Paris

[2]Sorbonne Université, Laboratoire d’informatique de Paris 6 (LIP6)

Weakly Consistent Behaviour

`x = 1;`	`y = 1;`
`r0 = y;`	`r1 = x;`

The Programmer's Intuition: Sequential Consistency
After execution, `r0 = 1 || r1 = 1`.

On x86
We can observe `r0 = 0 && r1 = 0`.

A More Useful Program: Peterson's Lock Algorithm

```
/* Thread 0 */
flag_0 = true;
// mfence;
turn = true;
mfence;

while (flag1 && turn) { }
critical_section_thread0:
flag_0 = false;
// mfence;
```

```
/* Thread 1 */
flag_1 = true;
// mfence;
turn = false;
mfence;

while (flag_0 && not turn) { }
critical_section_thread1:
flag_1 = false;
// mfence;
```

- Is mutual exclusion guaranteed ? Yes (previous work).
- Does the verification scale for N threads ?

A More Useful Program: Peterson's Lock Algorithm

```
/* Thread 0 */
flag_0 = true;
// mfence;
turn = true;
mfence;

while (flag1 && turn) { }
critical_section_thread0:
flag_0 = false;
// mfence;
```

```
/* Thread 1 */
flag_1 = true;
// mfence;
turn = false;
mfence;

while (flag_0 && not turn) { }
critical_section_thread1:
flag_1 = false;
// mfence;
```

- Is mutual exclusion guaranteed ? Yes (previous work).
- Does the verification scale for N threads ?

A More Useful Program: Peterson's Lock Algorithm

```
/* Thread 0 */
flag_0 = true;
// mfence;
turn = true;
mfence;

while (flag1 && turn) { }
critical_section_thread0:
flag_0 = false;
// mfence;
```

```
/* Thread 1 */
flag_1 = true;
// mfence;
turn = false;
mfence;

while (flag_0 && not turn) { }
critical_section_thread1:
flag_1 = false;
// mfence;
```

- Is mutual exclusion guaranteed ? Yes (previous work).
- Does the verification scale for N threads ?

A More Useful Program: Peterson's Lock Algorithm

```
/* Thread 0 */
flag_0 = true;
// mfence;
turn = true;
mfence;

while (flag1 && turn) { }
critical_section_thread0:
flag_0 = false;
// mfence;
```

```
/* Thread 1 */
flag_1 = true;
// mfence;
turn = false;
mfence;

while (flag_0 && not turn) { }
critical_section_thread1:
flag_1 = false;
// mfence;
```

- Is mutual exclusion guaranteed ? Yes (previous work).
- Does the verification scale for N threads ?

Presentation of the Problem

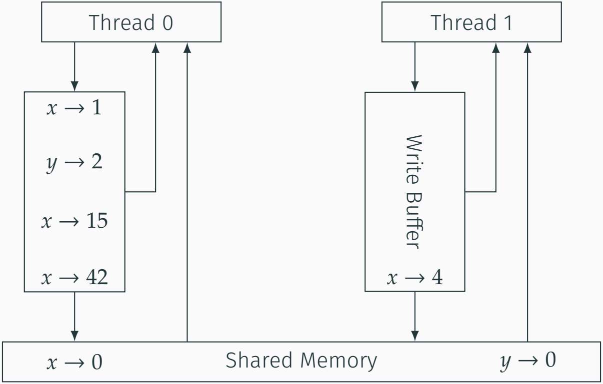
- We focus on verifying reachability properties on relaxed memory models.
- In a previous work, we used an abstract interpretation method based on array domains.
- We show how to extend it in a thread-modular way for scalability.

Plan

The Relaxed Memory Model

Total Store Ordering, the Base Model of x86

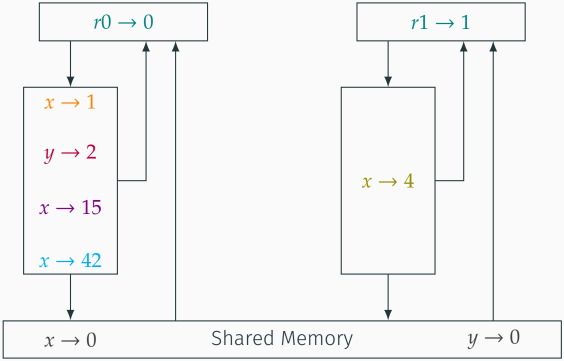
- Buffers are totally ordered FIFO queues.
- Buffer entries are flushed non-deterministically.
- Instruction `mfence` flushes the whole buffer of the thread.



Partial Store Ordering Encoding

We use one pseudo-variable for each buffer entry.

Inter-variables ordering is abstracted away.



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

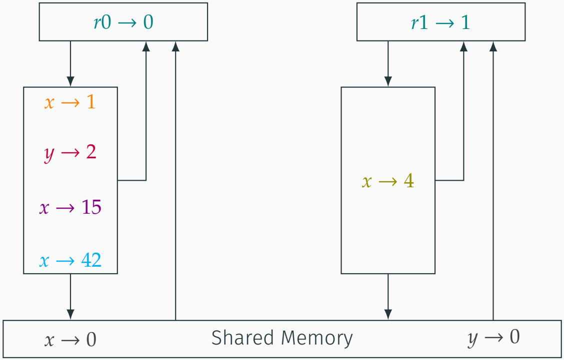
$x_2^0 = 15$

$x_3^0 = 42$

$x_1^1 = 4$

Partial Store Ordering Encoding

We use one pseudo-variable for each buffer entry.
Inter-variables ordering is abstracted away.



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

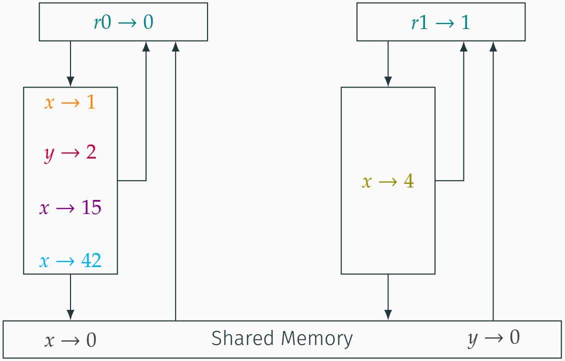
$x_2^0 = 15$

$x_3^0 = 42$

$x_1^1 = 4$

Partial Store Ordering Encoding

We use one pseudo-variable for each buffer entry.
Inter-variables ordering is abstracted away.



$$r0 = 0$$

$$r1 = 1$$

$$x_1^0 = 1$$

$$y_1^0 = 2$$

$$x_2^0 = 15$$

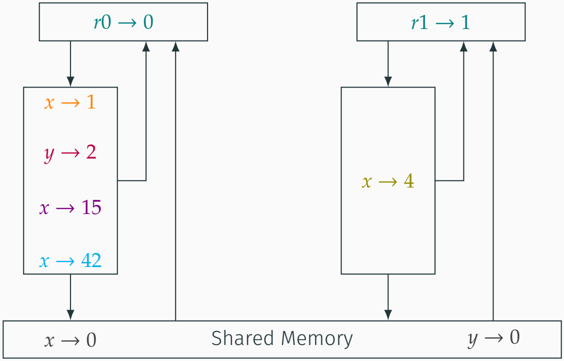
$$x_3^0 = 42$$

$$x_1^1 = 4$$

Partial Store Ordering Encoding

We use one pseudo-variable for each buffer entry.

Inter-variables ordering is abstracted away.



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

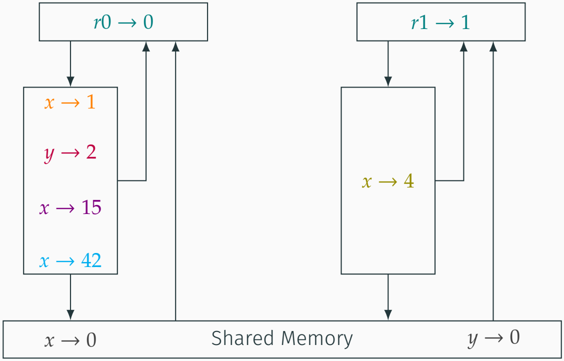
$x_2^0 = 15$

$x_3^0 = 42$

$x_1^1 = 4$

Partial Store Ordering Encoding

We use one pseudo-variable for each buffer entry.
Inter-variables ordering is abstracted away.



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

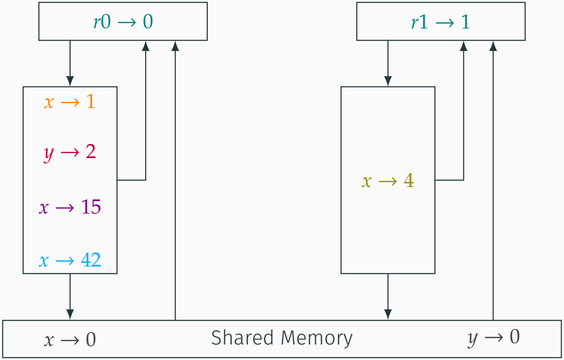
$x_2^0 = 15$

$x_3^0 = 42$

$x_1^1 = 4$

Partial Store Ordering Encoding

We use one pseudo-variable for each buffer entry.
Inter-variables ordering is abstracted away.



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

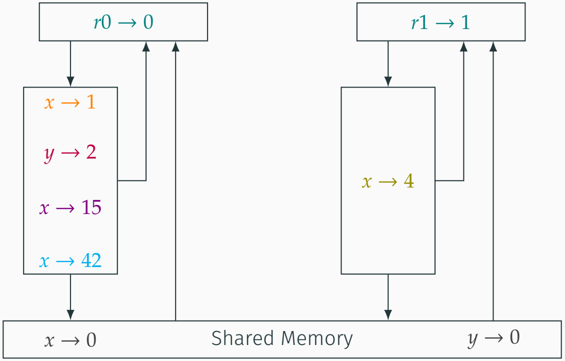
$x_2^0 = 15$

$x_3^0 = 42$

$x_1^1 = 4$

Partial Store Ordering Encoding

We use one pseudo-variable for each buffer entry.
Inter-variables ordering is abstracted away.



$$r0 = 0$$

$$r1 = 1$$

$$x_1^0 = 1$$

$$y_1^0 = 2$$

$$x_2^0 = 15$$

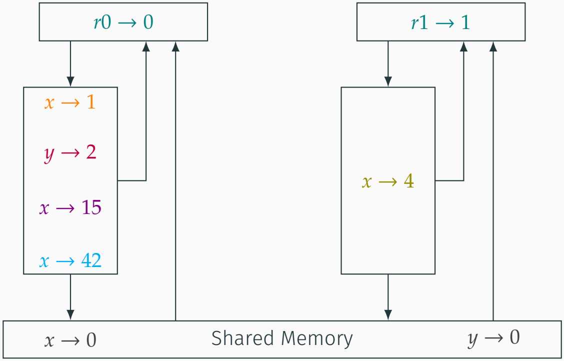
$$x_3^0 = 42$$

$$x_1^1 = 4$$

Partial Store Ordering Encoding

We use one pseudo-variable for each buffer entry.

Inter-variables ordering is abstracted away.



$$r0 = 0$$

$$r1 = 1$$

$$x_1^0 = 1$$

$$y_1^0 = 2$$

$$x_2^0 = 15$$

$$x_3^0 = 42$$

$$x_1^1 = 4$$

Monolithic Analysis

The Interleaving Graph

- We build the product control flow graph representing all possible interleavings.
- Each edge has a self loop `while(random) { flush oldest };` to represent the non-determinism of flushes.
- The analysis now becomes a usual fixpoint computation.

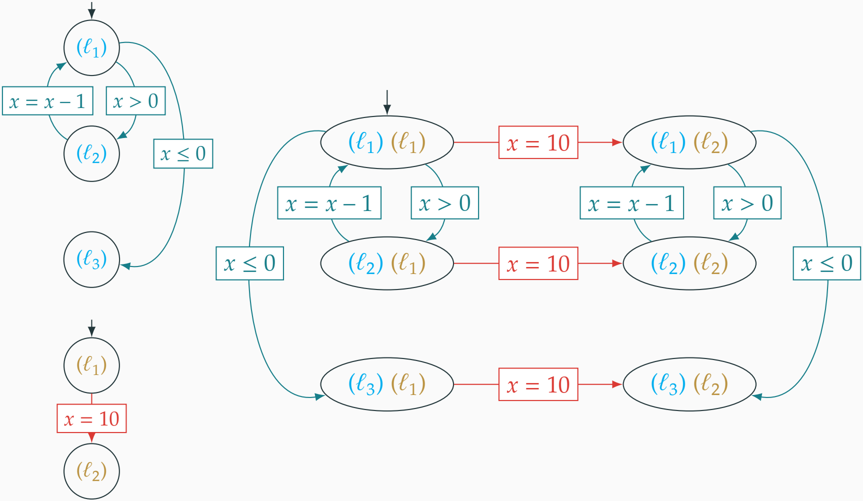
The Interleaving Graph

- We build the product control flow graph representing all possible interleavings.
- Each edge has a self loop `while(random) { flush oldest };` to represent the non-determinism of flushes.
- The analysis now becomes a usual fixpoint computation.

The Interleaving Graph

- We build the product control flow graph representing all possible interleavings.
- Each edge has a self loop `while(random) { flush oldest };` to represent the non-determinism of flushes.
- The analysis now becomes a usual fixpoint computation.

A product graph example



General Abstraction Idea

Buffers are the hard(est) part of the abstraction:

- They have an unbounded size.
- This size can change in a dynamic and undeterministic way even between execution steps.

We proposed to adapt array abstractions (specifically, summarisation[1]) to efficiently represent buffers.

[1] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps, and Mooly Sagiv. Numeric domains with summarized dimensions. In *TACAS 2004*.

Monolithic Analysis

Summarisation

Summarising the Buffers

Observation.

- The most recent entry, x_1^T, plays a special role: it will be used for reading x.
- The other entries are only destined to reach the memory eventually.

Summarisation.

- In each state where they are defined, we group the variables $x_2^T, \ldots, x_\infty^T$ into a single summarised variable x_{bot}^T.
- x_1^T is kept separated: otherwise, reading from the buffer would be very imprecise.

Summarising the Buffers

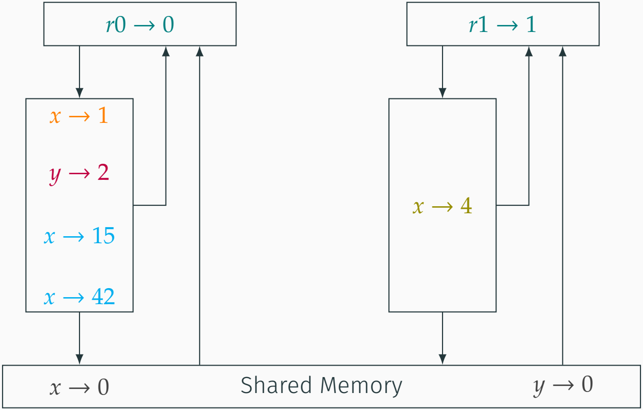
Observation.

- The most recent entry, x_1^T, plays a special role: it will be used for reading x.
- The other entries are only destined to reach the memory eventually.

Summarisation.

- In each state where they are defined, we group the variables $x_2^T, \dots, x_\infty^T$ into a single summarised variable x_{bot}^T.
- x_1^T is kept separated: otherwise, reading from the buffer would be very imprecise.

Summarisation: an Example



$$r0 = 0$$

$$r1 = 1$$

$$x_1^0 = 1$$

$$y_1^0 = 2$$

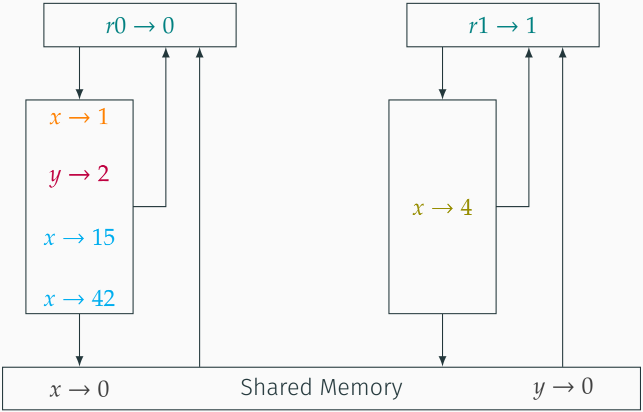
$$x_{bot}^0 \in \{15; 42\}$$

$$x_1^1 = 4$$

$$x^{mem} = 0$$

$$y^{mem} = 1$$

Summarisation: an Example



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

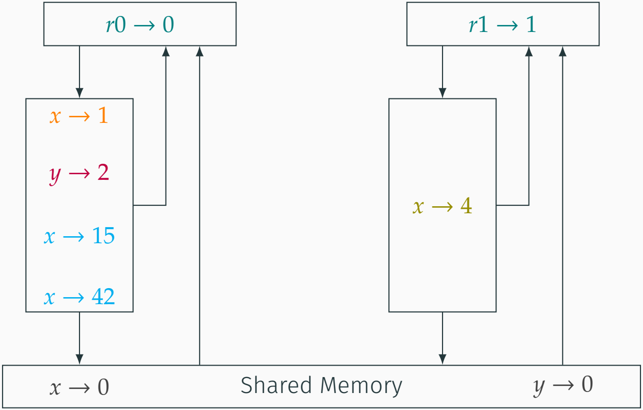
$x_{bot}^0 \in \{15; 42\}$

$x_1^1 = 4$

$x^{mem} = 0$

$y^{mem} = 1$

Summarisation: an Example



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

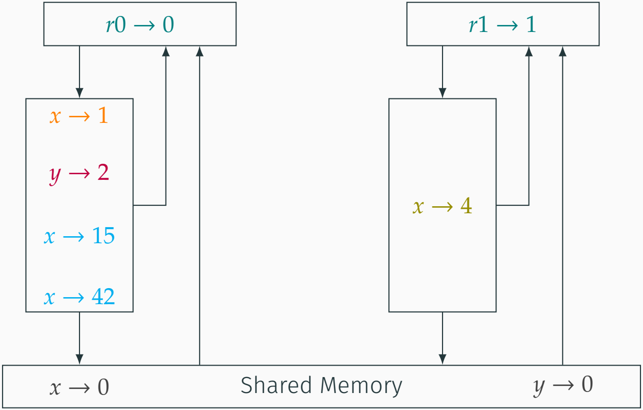
$x_{bot}^0 \in \{15; 42\}$

$x_1^1 = 4$

$x^{mem} = 0$

$y^{mem} = 1$

Summarisation: an Example



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

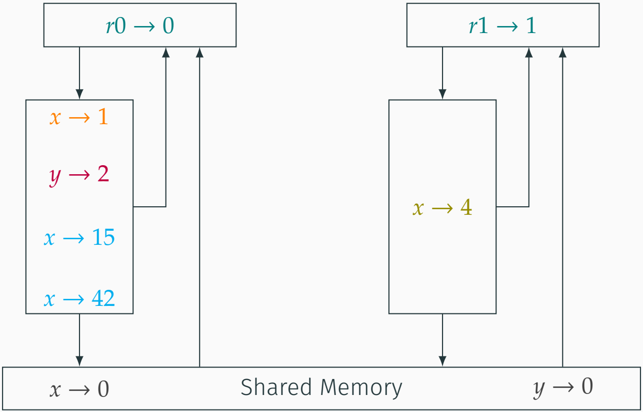
$x_{bot}^0 \in \{15; 42\}$

$x_1^1 = 4$

$x^{mem} = 0$

$y^{mem} = 1$

Summarisation: an Example



$r0 = 0$

$r1 = 1$

$x_1^0 = 1$

$y_1^0 = 2$

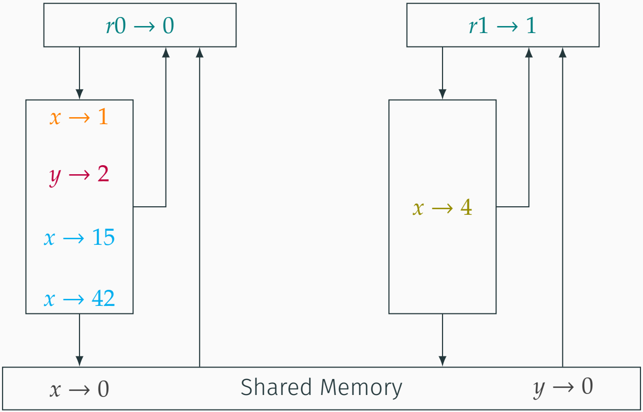
$x_{bot}^0 \in \{15; 42\}$

$x_1^1 = 4$

$x^{mem} = 0$

$y^{mem} = 1$

Summarisation: an Example



$$r0 = 0$$

$$r1 = 1$$

$$x_1^0 = 1$$

$$y_1^0 = 2$$

$$x_{bot}^0 \in \{15; 42\}$$

$$x_1^1 = 4$$

$$x^{mem} = 0$$

$$y^{mem} = 1$$

Monolithic Analysis

Final Abstraction

Partitioning and Numerical Domains for Abstraction

- Summarised states are partitioned according to the summarised variables they defined (equivalently, a partial information on buffer lengths).
- Then we can use numerical domains (octagons, polyhedra...) to abstract each partition.
- Partitioning also helps for the definition of abstract operators.

Modular analysis

A Scalability Issue

The interleavings analysis works well for 2 threads. What about... 3? 5? 10? 100?

$\implies$ Combinatorial explosion of the graph.

Monolithic analyses do not scale. We need thread modularity.

Modular analysis example

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

Modular analysis example

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0)	$\bot$
(ℓ_1)	$\bot$
(ℓ_2)	$\bot$

(ℓ_0)	$\bot$
(ℓ_1)	$\bot$
(ℓ_2)	$\bot$

Modular analysis example

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0)	$x = 0$
(ℓ_1)	$\perp$
(ℓ_2)	$\perp$

(ℓ_0)	$\perp$
(ℓ_1)	$\perp$
(ℓ_2)	$\perp$

Modular analysis example

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

T1:

(ℓ_0) $x = 0$

(ℓ_1) $\bot$

(ℓ_2) $\bot$

T2:

(ℓ_0) $x = 0$

(ℓ_1) $\bot$

(ℓ_2) $\bot$

Modular analysis example

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0)	$x = 0$
(ℓ_1)	$\bot$
(ℓ_2)	$\bot$

(ℓ_0)	$x = 0$
(ℓ_1)	$x = 0$
(ℓ_2)	$\bot$

Modular analysis example

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0)	$x = 0$
(ℓ_1)	$\perp$
(ℓ_2)	$\perp$

(ℓ_0)	$x = 0$
(ℓ_1)	$x = 0$
(ℓ_2)	$x = 1$

Modular analysis example

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0)	$x = 0$
(ℓ_1)	$\bot$
(ℓ_2)	$\bot$

(ℓ_0)	$x = 0$
(ℓ_1)	$x = 0$
(ℓ_2)	$x = 1$
Effect	$x \mapsto 1$

Modular analysis example

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0)	$x = 0$
(ℓ_1)	$\bot$
(ℓ_2)	$\bot$

(ℓ_0)	$x \in \{0, 1\}$
(ℓ_1)	$x = 0$
(ℓ_2)	$x = 1$
Effect	$x \mapsto 1$

Modular analysis example

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0)	$x = 0$
(ℓ_1)	$\bot$
(ℓ_2)	$\bot$

(ℓ_0)	$x \in \{0, 1\}$
(ℓ_1)	$x = 0$
(ℓ_2)	$x = 1$
Effect	$x \mapsto 1$

Modular analysis example

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0)	$x = 0$
(ℓ_1)	$\perp$
(ℓ_2)	$\perp$

(ℓ_0)	$x \in \{0, 1\}$
(ℓ_1)	$x = 0$
(ℓ_2)	$x = 1$
Effect	$x \mapsto 1$

Modular analysis example

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0)	$x = 0$
(ℓ_1)	$\bot$
(ℓ_2)	$\bot$

(ℓ_0)	$x \in \{0, 1\}$
(ℓ_1)	$x = 0$
(ℓ_2)	$x = 1$
Effect	$x \mapsto 1$

Modular analysis example

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0)	$x = 0$
(ℓ_1)	$\bot$
(ℓ_2)	$\bot$

(ℓ_0)	$x \in \{0, 1\}$
(ℓ_1)	$x = 0$
(ℓ_2)	$x = 1$
Effect	$x \mapsto 1$

Modular analysis example

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0)	$x = 0$
(ℓ_1)	$\perp$
(ℓ_2)	$\perp$

(ℓ_0)	$x \in \{0, 1\}$
(ℓ_1)	$x = 0$
(ℓ_2)	$x = 1$
Effect	$x \mapsto 1$

Modular analysis example

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0)	$x \in \{0, 1\}$
(ℓ_1)	$\bot$
(ℓ_2)	$\bot$

(ℓ_0)	$x \in \{0, 1\}$
(ℓ_1)	$x = 0$
(ℓ_2)	$x = 1$
Effect	$x \mapsto 1$

Modular analysis example

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0) $x \in \{0,1\}$
(ℓ_1) $x = 1$
(ℓ_2) $\bot$

(ℓ_0) $x \in \{0,1\}$
(ℓ_1) $x = 0$
(ℓ_2) $x = 1$
Effect $x \mapsto 1$

Interferences need only be applied at read points.

Modular analysis example

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0)	$x \in \{0, 1\}$
(ℓ_1)	$x = 1$
(ℓ_2)	$x = 0$
Effect	$x \mapsto 0$

(ℓ_0)	$x \in \{0, 1\}$
(ℓ_1)	$x = 0$
(ℓ_2)	$x = 1$
Effect	$x \mapsto 1$

Modular analysis example

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

(ℓ_0)	$x \in \{0, 1\}$
(ℓ_1)	$x = 1$
(ℓ_2)	$x = 0$
Effect	$x \mapsto 0$

(ℓ_0)	$x \in \{0, 1\}$
(ℓ_1)	$x = 0$
(ℓ_2)	$x = 1$
Effect	$x \mapsto 1$

Modular analysis

The Interferences Framework

Non-relational interferences

We just met them!

They are simple pairs (*variable* $\mapsto$ *possible new value*).

Relational interferences

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

T2 *may write* 1 *in* x *if* x *was previously equal to* 0*, and by doing so it would go from* (ℓ_1) *to* (ℓ_2).

Relational interferences...

- Link a variable modification to the previous state
- Hold control information

Relational interferences

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

T2 *may write* 1 *in* x *if* x *was previously equal to* 0, *and by doing so it would go from* (ℓ_1) *to* (ℓ_2).

Relational interferences...

- Link a variable modification to the previous state
- Hold control information

Relational interferences

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

T2 may write 1 in x if x was previously equal to 0, and by doing so it would go from (ℓ_1) to (ℓ_2).

Relational interferences...

- Link a variable modification to the previous state
- Hold control information

Relational interferences

```
thread /* T1 */ {
    while true {
        while (ℓ0) x != 1 {}
        /* Critical section ... */
        (ℓ1) x = 0; (ℓ2)
    }
}
```

```
thread /* T2 */ {
    while true {
        while (ℓ0) x != 0 {}
        /* Critical section ... */
        (ℓ1) x = 1; (ℓ2)
    }
}
```

T2 may write 1 in x if x was previously equal to 0, and by doing so it would go from (ℓ_1) to (ℓ_2).

Relational interferences...

- Link a variable modification to the previous state
- Hold control information

Summing up the interferences modular analysis

- Works as a nested fixpoint.
    - The inner fixpoint stabilises a thread result depending on a given interferences subset.
    - The outer fixpoint runs the inner fixpoint until stabilisation of the generated interferences.
- Global control information goes from structuring the analysis to being part of its state.
    - We add specific pc_T variables to thread states.

Modular analysis

Thread-Modular Abstractions

Local States of thread T

- We add an auxiliary variable $pc_{T'}$ for each $T' \neq T$
- We forget all buffers from other threads
- We still keep other threads local variables

Control Abstraction

Is the abstraction for the $pc_{T'}$ variables.

Mainly inspired by Raphaël Monat and Antoine Miné: Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. *VMCAI 2017.*

Possible choices:

- Flow insensitivity: $\alpha(\ell) = \top$.
- Concrete control: $\alpha(\ell) = \ell$.
- Control partitioning: we group together chosen control points.

Control partitioning: splitting at property points

```
thread /* T1 */ {
    (ℓ0) while (ℓ1) true {
        (ℓ2) while (ℓ3) x != 1 {(ℓ4)} (ℓ5)
        /* Critical section ... */
        (ℓ6) x = 0; (ℓ7)
    } (ℓ8)
} (ℓ9)
```

$$\alpha((\ell_1)..(\ell_4)) = \ell_1^\sharp$$

$$\alpha((\ell_5)..(\ell_8)) = \ell_2^\sharp$$

Control partitioning: also splitting at loop heads

```
thread /* T1 */ {
    (ℓ0) while (ℓ1) true {
        (ℓ2) while (ℓ3) x != 1 {(ℓ4)} (ℓ5)
        /* Critical section ... */
        (ℓ6) x = 0; (ℓ7)
    } (ℓ8)
} (ℓ9)
```

$$\alpha((\ell_0)) = \ell_0^\sharp$$
$$\alpha((\ell_1)..(\ell_2)) = \ell_1^\sharp$$
$$\alpha((\ell_3)..(\ell_4)) = \ell_2^\sharp$$
$$\alpha((\ell_5)..(\ell_9)) = \ell_3^\sharp$$

Interferences

They are built with the “least common information” of local states abstractions:

- We forget every buffer
- We keep each pc_T variable to represent control

To represent pairs of states, we used “primed” variables:

$$x = 0, y = 1, pc_1 = (\ell_0), pc_2 = (\ell_2), x' = 1, y' = 1, pc'_1 = (\ell_2), pc'_2 = (\ell_2)$$

Then we apply the same control and numerical abstraction as for the local states.

Flush closure: an optimisation

Flush is non-deterministic: we need *closed-by-flush* results.

Naive flush closure

- At each step, we flush everything until we reach a fixpoint.
- Flushes discover new applicable interferences, and we need to close after interference application.

"Smart" flush closure

- $[\![flush\ z]\!] \circ [\![x \leftarrow 1]\!] = [\![x \leftarrow 1]\!] \circ [\![flush\ z]\!]$
- From a closed element, it is sufficient to compute the flushes of x after operations that read or write x.
- We label and partition the interferences by the shared variable they are related to.

Flush closure: an optimisation

Flush is non-deterministic: we need *closed-by-flush* results.

Naive flush closure

- At each step, we flush everything until we reach a fixpoint.
- Flushes discover new applicable interferences, and we need to close after interference application.

"Smart" flush closure

- $[\![flush\ z]\!] \circ [\![x \leftarrow 1]\!] = [\![x \leftarrow 1]\!] \circ [\![flush\ z]\!]$
- From a closed element, it is sufficient to compute the flushes of x after operations that read or write x.
- We label and partition the interferences by the shared variable they are related to.

Flush closure: an optimisation

Flush is non-deterministic: we need *closed-by-flush* results.

Naive flush closure

- At each step, we flush everything until we reach a fixpoint.
- Flushes discover new applicable interferences, and we need to close after interference application.

"Smart" flush closure

- $[\![flush\ z]\!] \circ [\![x \leftarrow 1]\!] = [\![x \leftarrow 1]\!] \circ [\![flush\ z]\!]$
- From a closed element, it is sufficient to compute the flushes of x after operations that read or write x.
- We label and partition the interferences by the shared variable they are related to.

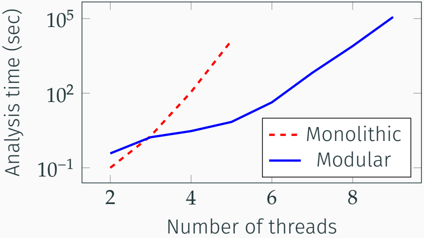
Results

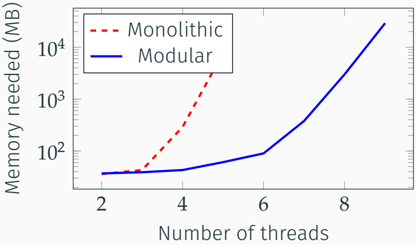
Precision

Small “hard to check” examples (typically mutual exclusion algorithms).

Test	Monolithic	Modular
abp	✓	✓
concloop	✓	✓
kessel	✓	✗
dekker	✓	✓
peterson	✓	✓
queue	✓	✓
bakery	⌛	✗

Scalability






```
thread {
  while (x != 0) { };
  x = 1;
}

thread {
  while (x != 1) { };
  x = 2;
}
/* .......... */
thread {
  while (x != N) { };
  x = 0;
}
```

Conclusion

- In a previous work: abstract interpretation under relaxed memory models.
- We show how to extend it in a thread-modular way.
- We got encouraging results : similar precision, better scaling.
- Future work:
  - Other models
  - Production-grade scaling †
  - We went from 2 to 10, can we go from 10 to N? †

† Thread-modularity is a prerequisite!

Thanks for your attention !