Effective Lock Handling in Stateless Model Checking

Michalis Kokologiannakis Azalea Raad Viktor Vafeiadis

October 25, 2019

Max Planck Institute for Software Systems (MPI-SWS)

Why Locks?

Why Locks?

(Coarse-grained) locking is the most widespread form of synchronization

- Programming languages provide syntactic support for it (e.g., Java)
- Modern hardware provides lock elision support

Why Locks?

(Coarse-grained) locking is the most widespread form of synchronization

- Programming languages provide syntactic support for it (e.g., Java)
- Modern hardware provides lock elision support

Verifying programs with locks is hard!

Verifying Programs With Locks Is Hard

Verifying Programs With Locks Is Hard

Consider a **fine-grained** hashtable

Let us try to verify it using a state-of-the-art **stateless model checking** (SMC) tool

Verifying Programs With Locks Is Hard

Consider a **fine-grained** hashtable

Let us try to verify it using a state-of-the-art **stateless model checking** (SMC) tool

#Threads	Fine-grained HT
5	0.05 s
6	0.05 s
7	0.06 s
8	0.08 s

Verification time using GenMC

Verifying Programs With Locks Is Hard

Consider a **fine-grained** hashtable

Let us try to verify it using a state-of-the-art **stateless model checking** (SMC) tool

#Threads	Fine-grained HT	Coarse-grained HT
5	0.05 s	104 s
6	0.05 s	⏰
7	0.06 s	⏰
8	0.08 s	⏰

Verification time using GenMC

Why Is Verifying Programs With Locks Hard?

Why Is Verifying Programs With Locks Hard?

SMC enumerates all the interleavings of a program

$\rightsquigarrow$ it is usually combined with **Partial Order Reduction (POR)**

Why Is Verifying Programs With Locks Hard?

SMC enumerates all the interleavings of a program

$\rightsquigarrow$ it is usually combined with **Partial Order Reduction (POR)**

$[x = 0]$		
$a := x;$	$\|\|$	$y := 42;$

Executions:

Why Is Verifying Programs With Locks Hard?

SMC enumerates all the interleavings of a program

$\rightsquigarrow$ it is usually combined with **Partial Order Reduction (POR)**

$[x = 0]$	
$a := x;$	$y := 42;$

Executions:

SMC-naive : 2

Why Is Verifying Programs With Locks Hard?

SMC enumerates all the interleavings of a program

$\rightsquigarrow$ it is usually combined with **Partial Order Reduction (POR)**

$[x = 0]$		
$a := x;$	$\|\|$	$y := 42;$

Executions:

SMC-naive : 2

SMC+POR : 1

Why Is Verifying Programs With Locks Hard?

SMC enumerates all the interleavings of a program

$\rightsquigarrow$ it is usually combined with **Partial Order Reduction (POR)**

$[x = 0]$	$[l = 0]$
$a := x; \;\|\|\; y := 42;$	$lock(l); \;\|\|\; lock(l);$

Executions:

SMC-naive : 2

SMC+POR : 1

Executions:

Why Is Verifying Programs With Locks Hard?

SMC enumerates all the interleavings of a program

$\rightsquigarrow$ it is usually combined with **Partial Order Reduction (POR)**

$[x = 0]$	$[l = 0]$
$a := x; \parallel y := 42;$	$lock(l); \parallel lock(l);$

Executions:

SMC-naive : 2

SMC+POR : 1

Executions:

SMC-naive : 2

Why Is Verifying Programs With Locks Hard?

SMC enumerates all the interleavings of a program

$\rightsquigarrow$ it is usually combined with **Partial Order Reduction (POR)**

$[x = 0]$	$[l = 0]$
$a := x; \;\Big\|\; y := 42;$	$lock(l); \;\Big\|\; lock(l);$

Executions:

SMC-naive : 2

SMC+POR : 1

Executions:

SMC-naive : 2

SMC+POR : 2

Why Is Verifying Programs With Locks Hard?

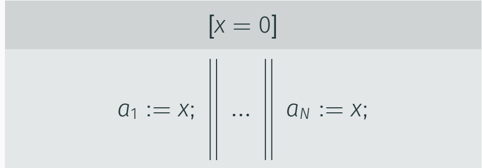
Why Is Verifying Programs With Locks Hard?

$$[x = 0]$$

$$a_1 := x; \;\Big\|\; \ldots \;\Big\|\; a_N := x;$$

Executions:

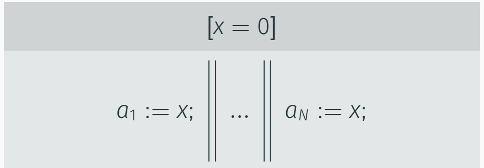
Why Is Verifying Programs With Locks Hard?



Executions:

SMC-naive : N!

Why Is Verifying Programs With Locks Hard?



Executions:

SMC-naive : N!

SMC+POR : 1

Why Is Verifying Programs With Locks Hard?

$[x = 0]$
$a_1 := x; \parallel \ldots \parallel a_N := x;$

$[x = 0]$		
$lock(l);$ $a_1 := x;$ $unlock(l);$	$\parallel \ldots \parallel$	$lock(l);$ $a_N := x;$ $unlock(l);$

Executions:

SMC-naive : N!

SMC+POR : 1

Executions:

Why Is Verifying Programs With Locks Hard?

$[x = 0]$
$a_1 := x; \;\Big\|\; \ldots \;\Big\|\; a_N := x;$

$[x = 0]$		
$lock(l);$ $a_1 := x;$ $unlock(l);$	$\ldots$	$lock(l);$ $a_N := x;$ $unlock(l);$

Executions:

SMC-naive : N!

SMC+POR : 1

Executions:

SMC-naive : N!

Why Is Verifying Programs With Locks Hard?

$[x = 0]$		
$a_1 := x;$	...	$a_N := x;$

$[x = 0]$		
$lock(l);$ $a_1 := x;$ $unlock(l);$	...	$lock(l);$ $a_N := x;$ $unlock(l);$

Executions:
SMC-naive : N!
SMC+POR : 1

Executions:
SMC-naive : N!
SMC+POR : N!

Why Is Verifying Programs With Locks Hard?

$[x = 0]$		
$a_1 := x;$	$\ldots$	$a_N := x;$

$[x = 0]$		
$lock(l);$ $a_1 := x;$ $unlock(l);$	$\ldots$	$lock(l);$ $a_N := x;$ $unlock(l);$

Executions:
SMC-naive : N!
SMC+POR : 1

Executions:
SMC-naive : N!
SMC+POR : N!

Locks inhibit Partial Order Reduction (POR)

Our Contribution

Our Contribution

- LAPOR: a **Lock-Aware Partial Order Reduction** algorithm
  - independence at both instruction and critical-section level

Our Contribution

- LAPOR: a **Lock-Aware Partial Order Reduction** algorithm
  - independence at both instruction and critical-section level
- We prove that LAPOR is sound, complete, optimal

Our Contribution

- LAPOR: a **Lock-Aware Partial Order Reduction** algorithm
  - independence at both instruction and critical-section level
- We prove that LAPOR is sound, complete, optimal
- Built on top of the GenMC framework
  - Works for a variety of memory models

Our Contribution

- LAPOR: a **Lock-Aware Partial Order Reduction** algorithm
  - independence at both instruction and critical-section level
- We prove that LAPOR is sound, complete, optimal
- Built on top of the GenMC framework
  - Works for a variety of memory models
- Exponentially fewer executions than state-of-the-art

Partial Order Reduction

SMC + POR: Background

Equivalent interleavings are represented by graphs:

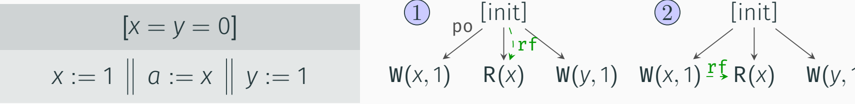
SMC + POR: Background

Equivalent interleavings are represented by graphs:

$[x = y = 0]$
$x := 1 \;\Big\|\; a := x \;\Big\|\; y := 1$

SMC + POR: Background

Equivalent interleavings are represented by graphs:



SMC + POR: Background

Equivalent interleavings are represented by graphs:

$[x = y = 0]$
$x := 1 \parallel a := x \parallel y := 1$

① [init] po rf W(x, 1) R(x) W(y, 1)

② [init] W(x, 1) rf R(x) W(y, 1)

Goal: Enumerate all consistent execution graphs of a program

SMC+POR: An Example

SMC+POR: An Example

$[x = y = 0]$
$x := 1 \parallel a := x \parallel y := 1$

SMC+POR: An Example

$[x = y = 0]$
$x := 1 \parallel a := x \parallel y := 1$

[init]

SMC+POR: An Example

$[x = y = 0]$
$x := 1 \parallel a := x \parallel y := 1$

[init]

$\mathtt{W}(x, 1)$

SMC+POR: An Example

$[x = y = 0]$
$x := 1 \parallel a := x \parallel y := 1$

$[\text{init}]$

$\mathtt{W}(x, 1) \xrightarrow{\mathtt{rf}} \mathtt{R}(x)$

SMC+POR: An Example

$[x = y = 0]$
$x := 1 \parallel a := x \parallel y := 1$

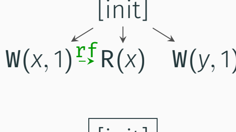
[init]

$\mathtt{W}(x, 1) \xrightarrow{\mathtt{rf}} \mathtt{R}(x)$

[init]

SMC+POR: An Example

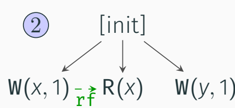
$[x = y = 0]$
$x := 1 \parallel a := x \parallel y := 1$

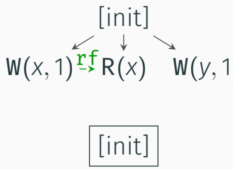


SMC+POR: An Example

$$[x = y = 0]$$

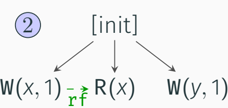
$$x := 1 \;\Big\|\; a := x \;\Big\|\; y := 1$$

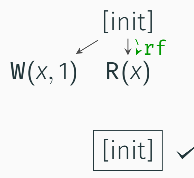




SMC+POR: An Example

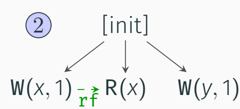
$[x = y = 0]$
$x := 1 \parallel a := x \parallel y := 1$

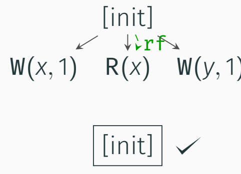




SMC+POR: An Example

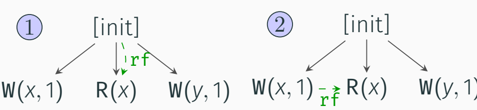
$[x = y = 0]$
$x := 1 \parallel a := x \parallel y := 1$

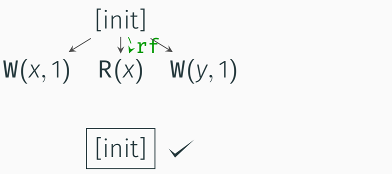




SMC+POR: An Example

$[x = y = 0]$
$x := 1 \parallel a := x \parallel y := 1$



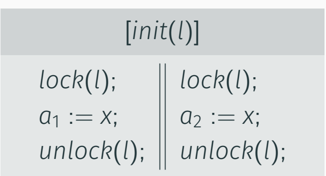


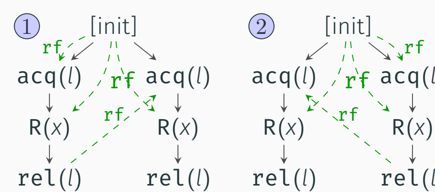
SMC+POR: Handling Locks

SMC+POR: Handling Locks

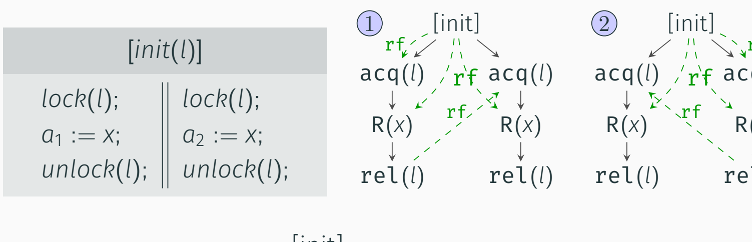
$[init(l)]$	
$lock(l);$	$lock(l);$
$a_1 := x;$	$a_2 := x;$
$unlock(l);$	$unlock(l);$

SMC+POR: Handling Locks





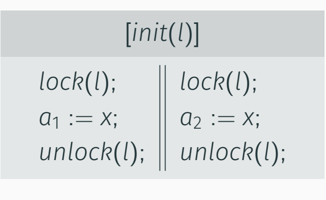
SMC+POR: Handling Locks

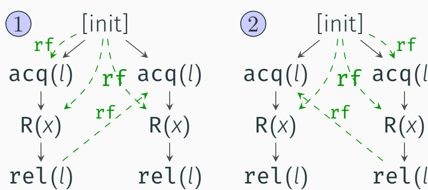


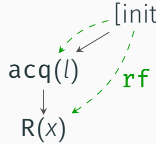
[init]

acq(l)

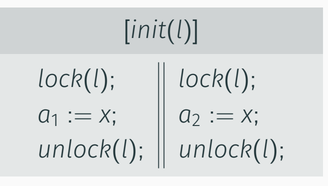
SMC+POR: Handling Locks

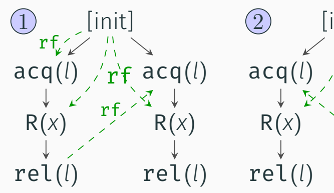


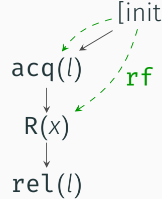




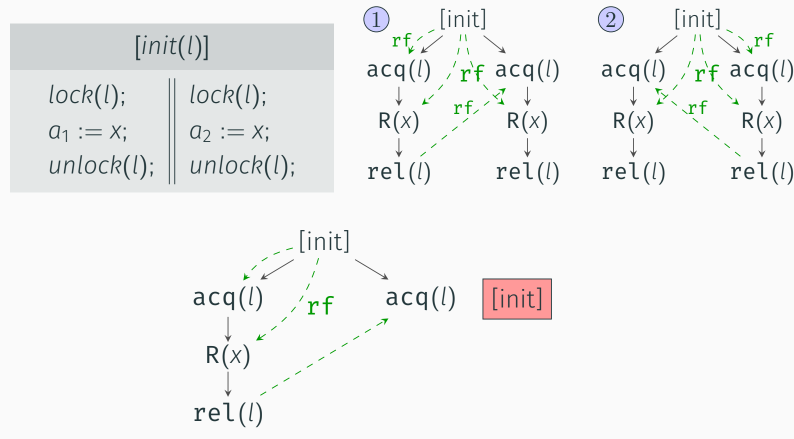
SMC+POR: Handling Locks



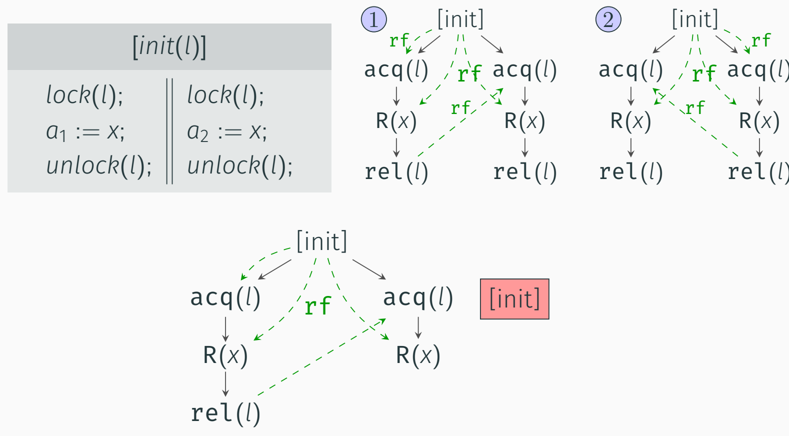




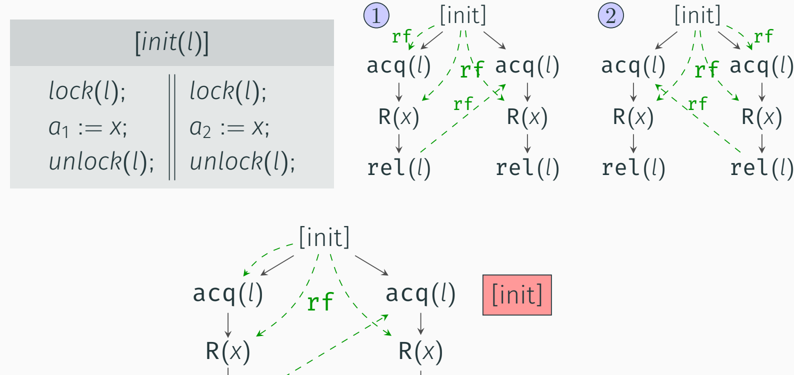
SMC+POR: Handling Locks



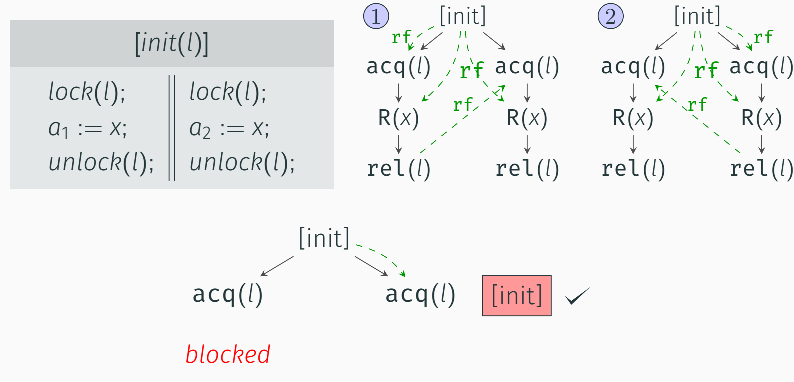
SMC+POR: Handling Locks



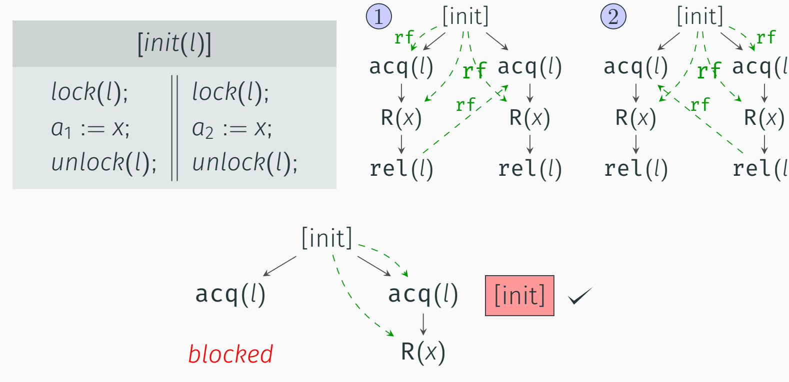
SMC+POR: Handling Locks



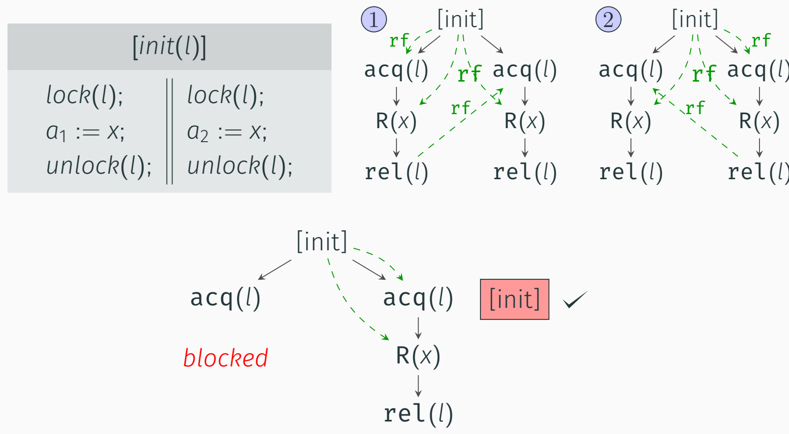
SMC+POR: Handling Locks



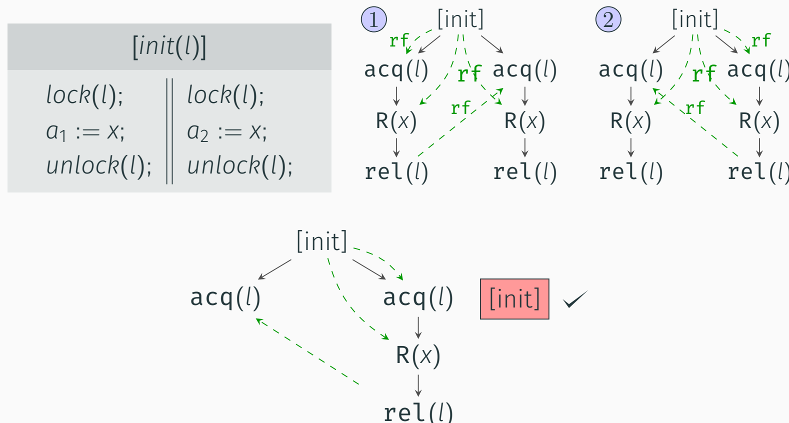
SMC+POR: Handling Locks



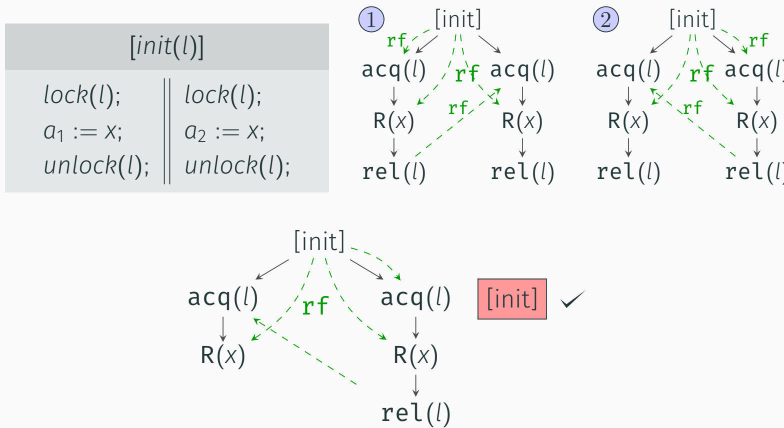
SMC+POR: Handling Locks



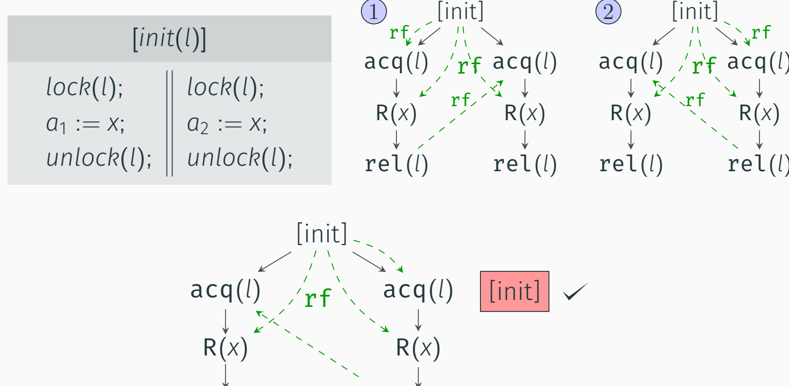
SMC+POR: Handling Locks



SMC+POR: Handling Locks



SMC+POR: Handling Locks



Lock-Aware Partial Order Reduction

LAPOR: Keeping Locks Unordered

$[init(l)]$	
$lock(l);$	$lock(l);$
$a_1 := x;$	$a_2 := x;$
$unlock(l);$	$unlock(l);$

LAPOR: Keeping Locks Unordered

$[init(l)]$	
$lock(l);$	$lock(l);$
$a_1 := x;$	$a_2 := x;$
$unlock(l);$	$unlock(l);$

(1)

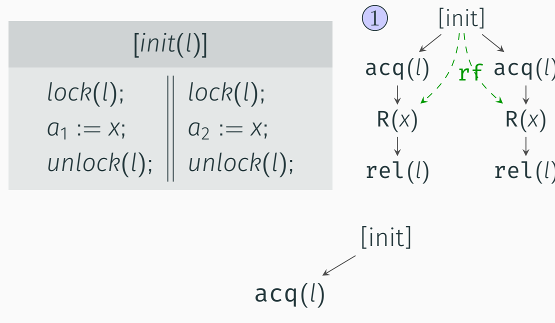
[init]

acq(l) rf acq(l)

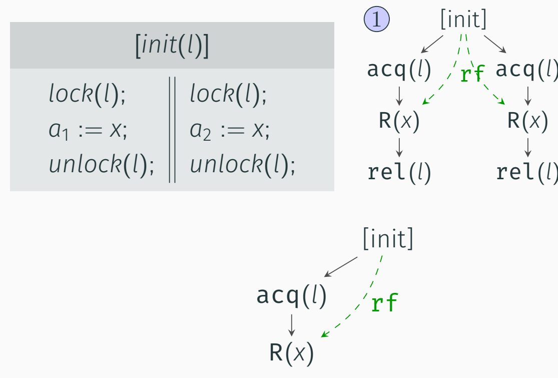
R(x) R(x)

rel(l) rel(l)

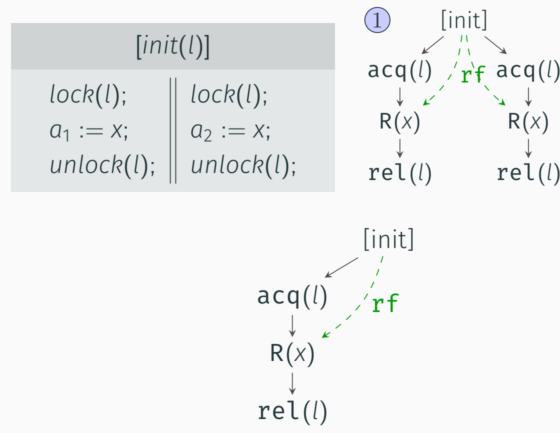
LAPOR: Keeping Locks Unordered



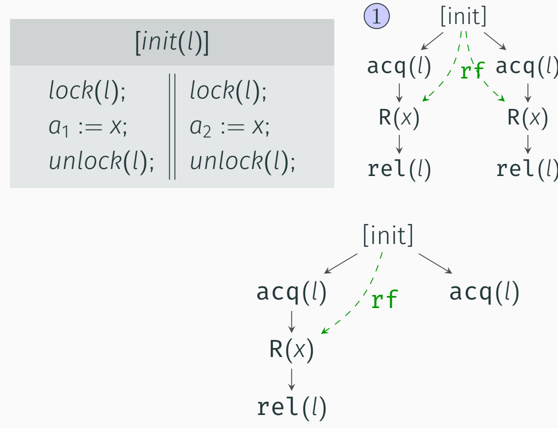
LAPOR: Keeping Locks Unordered



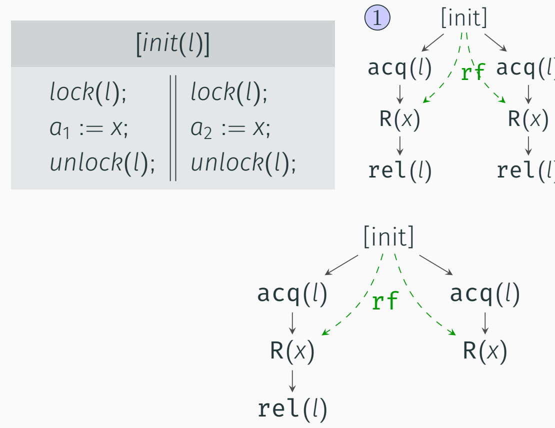
LAPOR: Keeping Locks Unordered



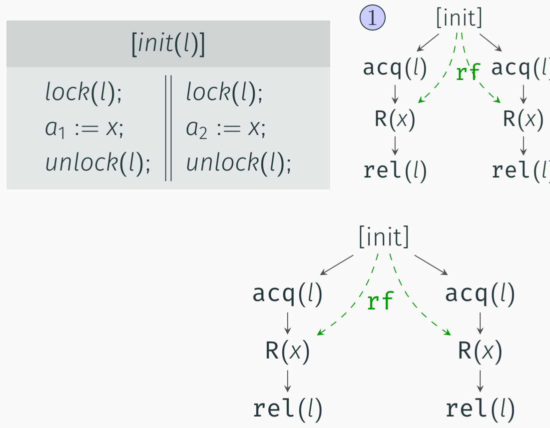
LAPOR: Keeping Locks Unordered



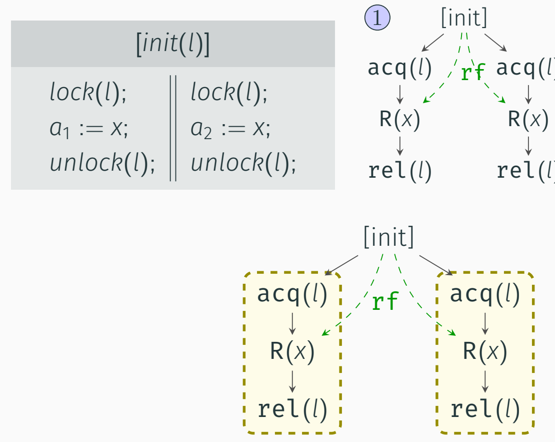
LAPOR: Keeping Locks Unordered



LAPOR: Keeping Locks Unordered



LAPOR: Keeping Locks Unordered

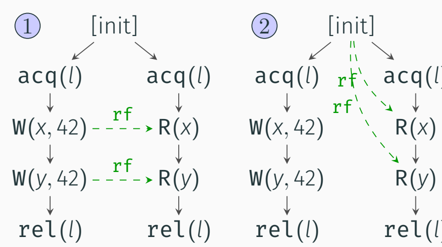


LAPOR: Keeping Locks Unordered

$[init(l)]$	
$lock(l);$	$lock(l);$
$x := 42;$	$a := x;$
$y := 42;$	$b := y;$
$unlock(l);$	$unlock(l);$

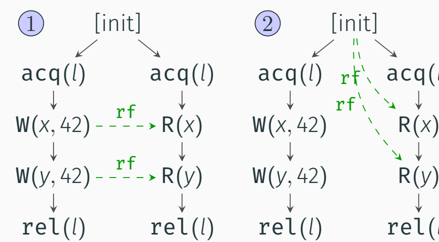
LAPOR: Keeping Locks Unordered

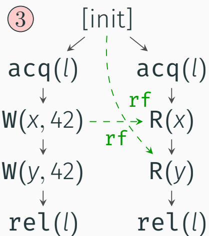
$[init(l)]$	
$lock(l);$	$lock(l);$
$x := 42;$	$a := x;$
$y := 42;$	$b := y;$
$unlock(l);$	$unlock(l);$



LAPOR: Keeping Locks Unordered

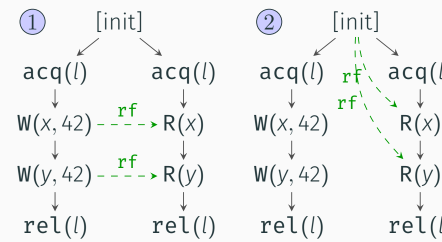
$[init(l)]$	
$lock(l);$	$lock(l);$
$x := 42;$	$a := x;$
$y := 42;$	$b := y;$
$unlock(l);$	$unlock(l);$

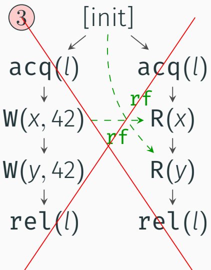




LAPOR: Keeping Locks Unordered

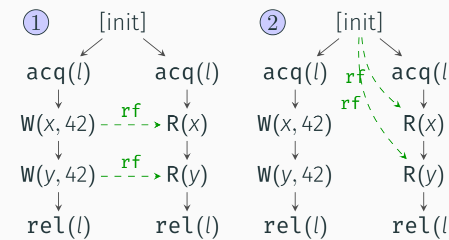
$[init(l)]$	
$lock(l);$	$lock(l);$
$x := 42;$	$a := x;$
$y := 42;$	$b := y;$
$unlock(l);$	$unlock(l);$

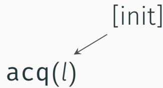




LAPOR: Keeping Locks Unordered

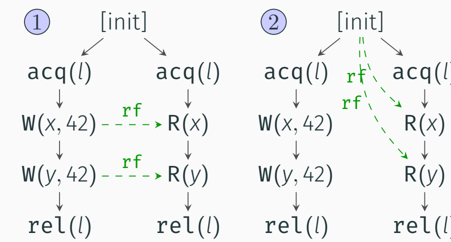
[*init*(*l*)]	
lock(*l*);	*lock*(*l*);
$x := 42$;	$a := x$;
$y := 42$;	$b := y$;
unlock(*l*);	*unlock*(*l*);

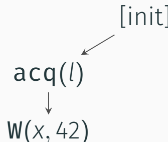




LAPOR: Keeping Locks Unordered

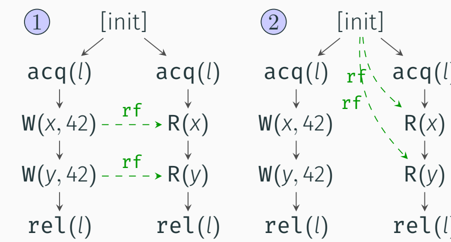
[$init(l)$]	
$lock(l);$	$lock(l);$
$x := 42;$	$a := x;$
$y := 42;$	$b := y;$
$unlock(l);$	$unlock(l);$

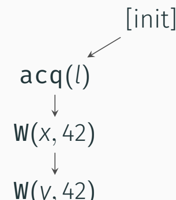




LAPOR: Keeping Locks Unordered

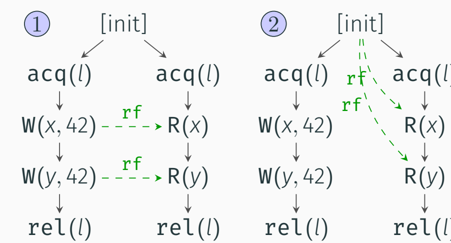
[init(l)]	
lock(l);	*lock(l);*
$x := 42;$	$a := x;$
$y := 42;$	$b := y;$
unlock(l);	*unlock(l);*

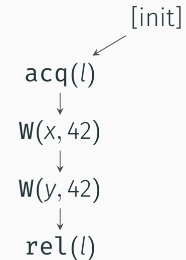




LAPOR: Keeping Locks Unordered

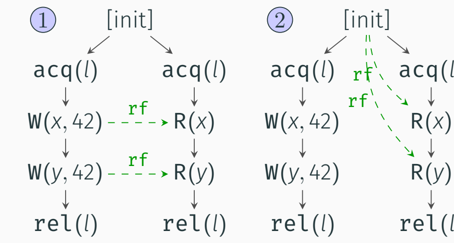
[*init*(*l*)]	
lock(*l*);	*lock*(*l*);
$x := 42$;	$a := x$;
$y := 42$;	$b := y$;
unlock(*l*);	*unlock*(*l*);

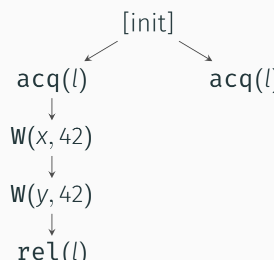




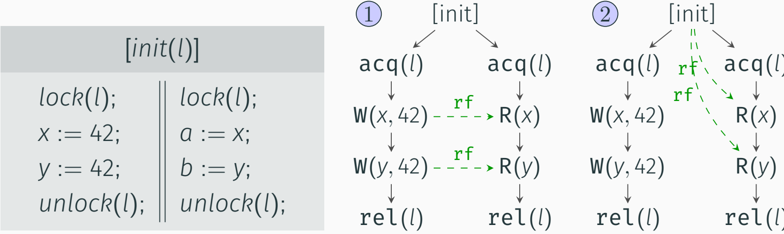
LAPOR: Keeping Locks Unordered

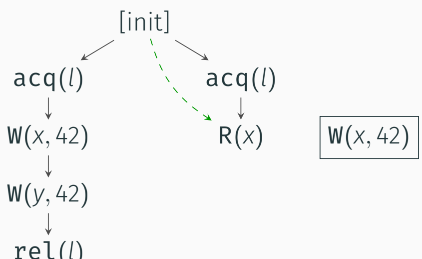
[*init*(*l*)]	
lock(*l*);	*lock*(*l*);
$x := 42$;	$a := x$;
$y := 42$;	$b := y$;
unlock(*l*);	*unlock*(*l*);





LAPOR: Keeping Locks Unordered





LAPOR: Keeping Locks Unordered

$[init(l)]$	
$lock(l);$	$lock(l);$
$x := 42;$	$a := x;$
$y := 42;$	$b := y;$
$unlock(l);$	$unlock(l);$

① [init]

acq(l) → W(x, 42) → W(y, 42) → rel(l)

acq(l) → R(x) → R(y) → rel(l)

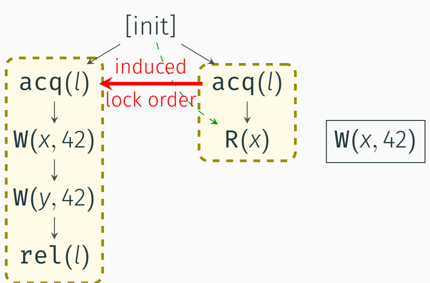
W(x, 42) --rf--> R(x); W(y, 42) --rf--> R(y)

② [init]

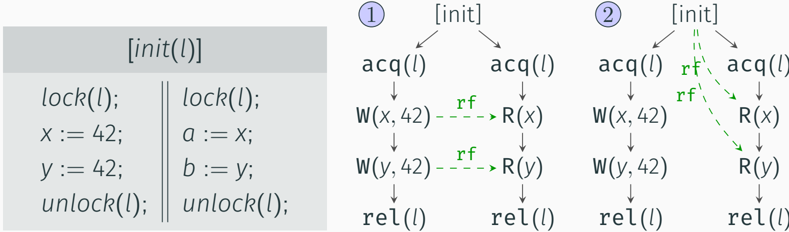
acq(l) → W(x, 42) → W(y, 42) → rel(l)

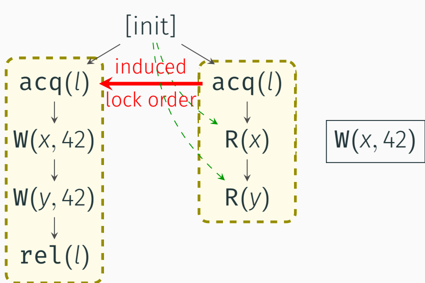
acq(l) → R(x) → R(y) → rel(l)

[init] --rf--> R(x); [init] --rf--> R(y)

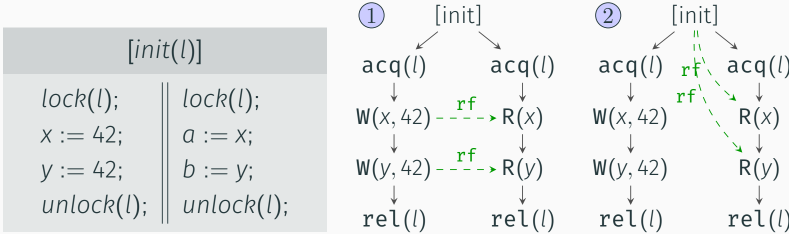


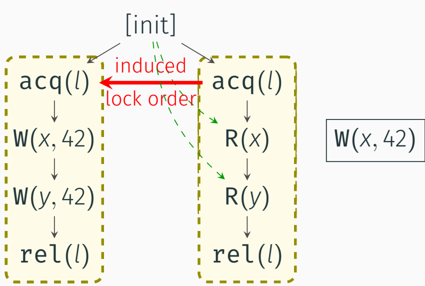
LAPOR: Keeping Locks Unordered





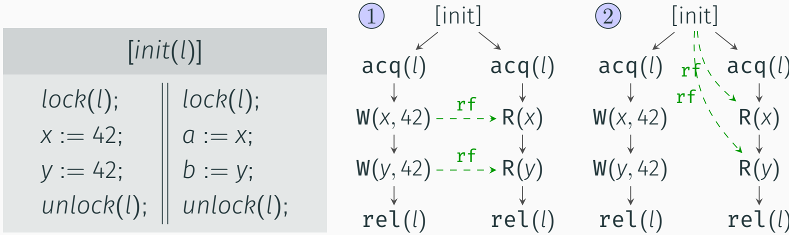
LAPOR: Keeping Locks Unordered

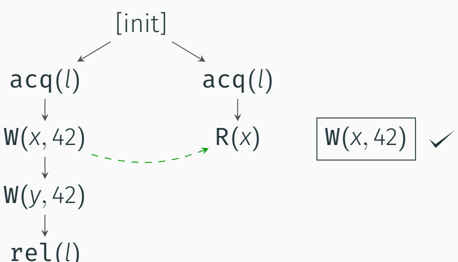




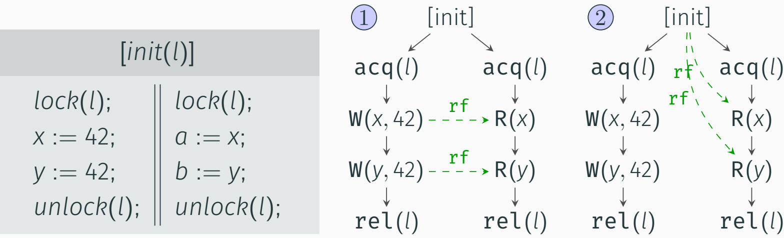
W(x, 42)

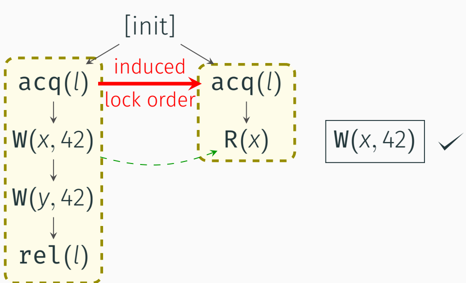
LAPOR: Keeping Locks Unordered



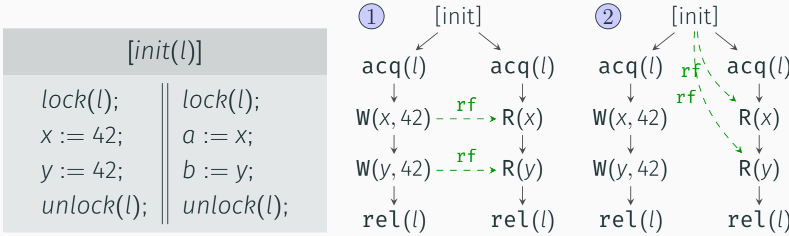


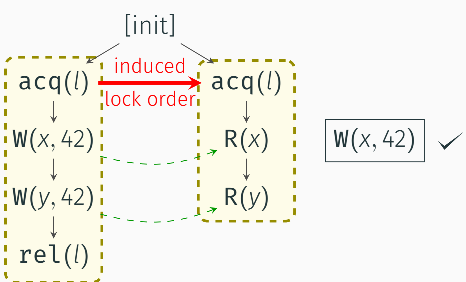
LAPOR: Keeping Locks Unordered



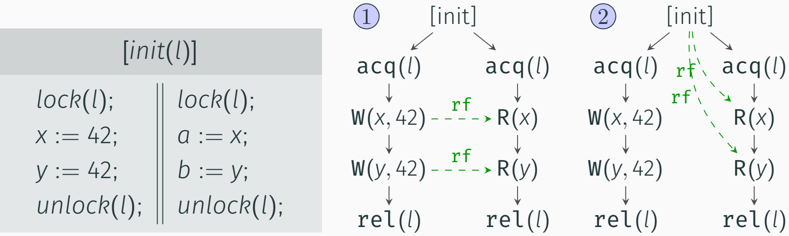


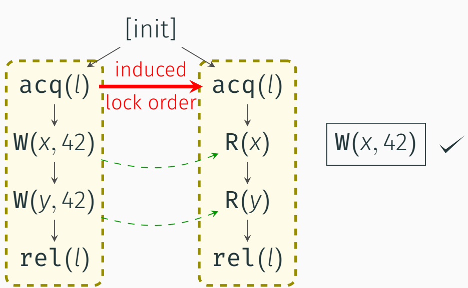
LAPOR: Keeping Locks Unordered





LAPOR: Keeping Locks Unordered

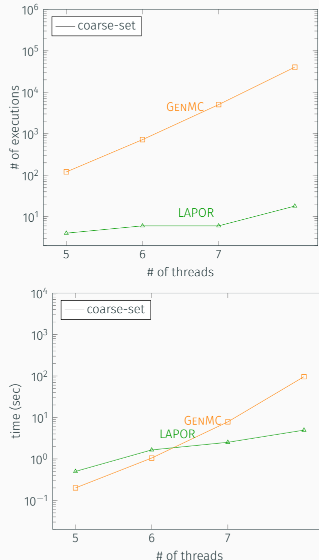




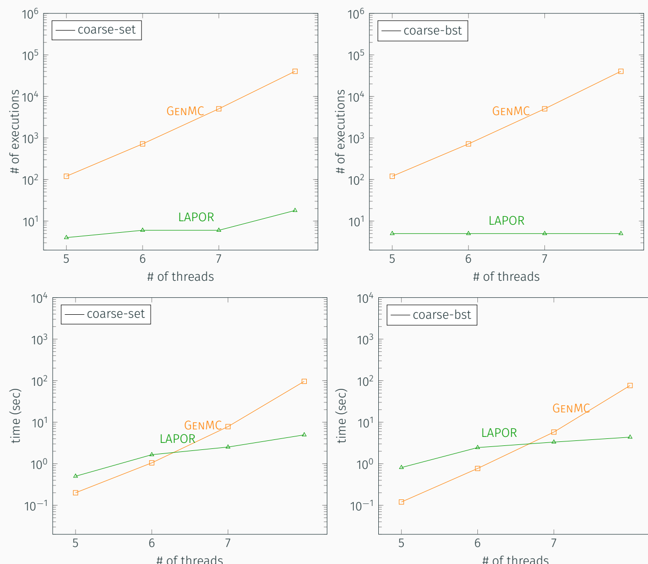
W(x, 42) ✓

Results

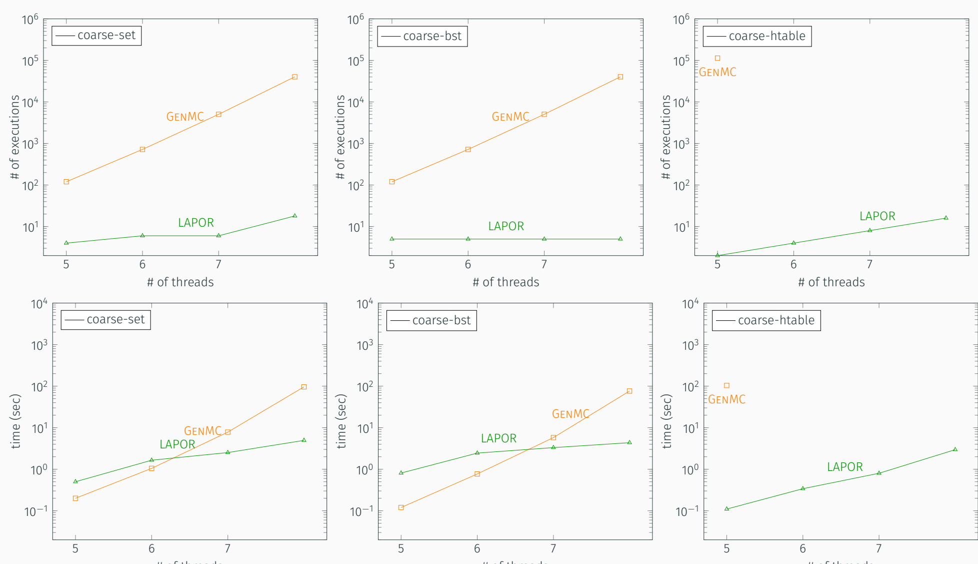
Coarse-Grained Data-Structures



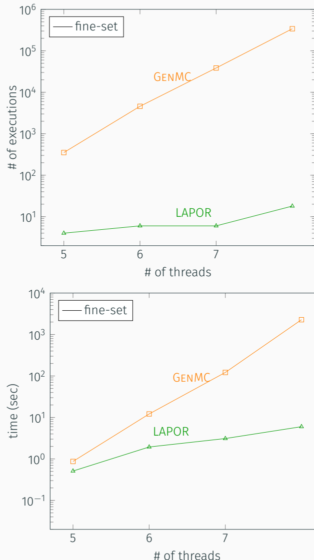
Coarse-Grained Data-Structures



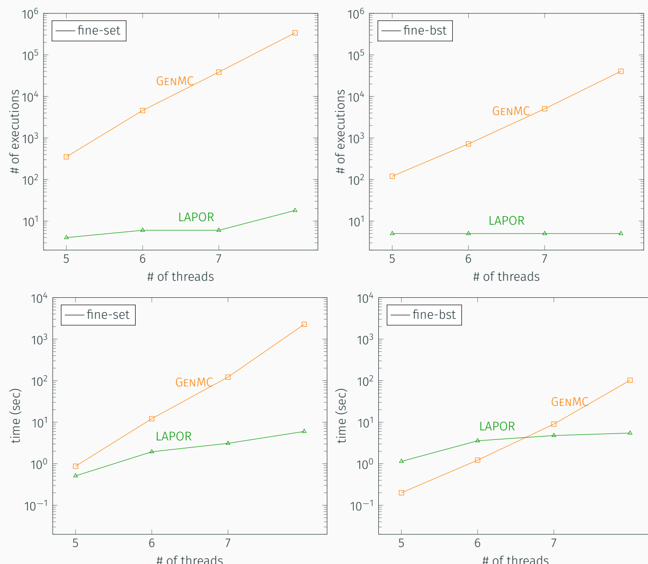
Coarse-Grained Data-Structures



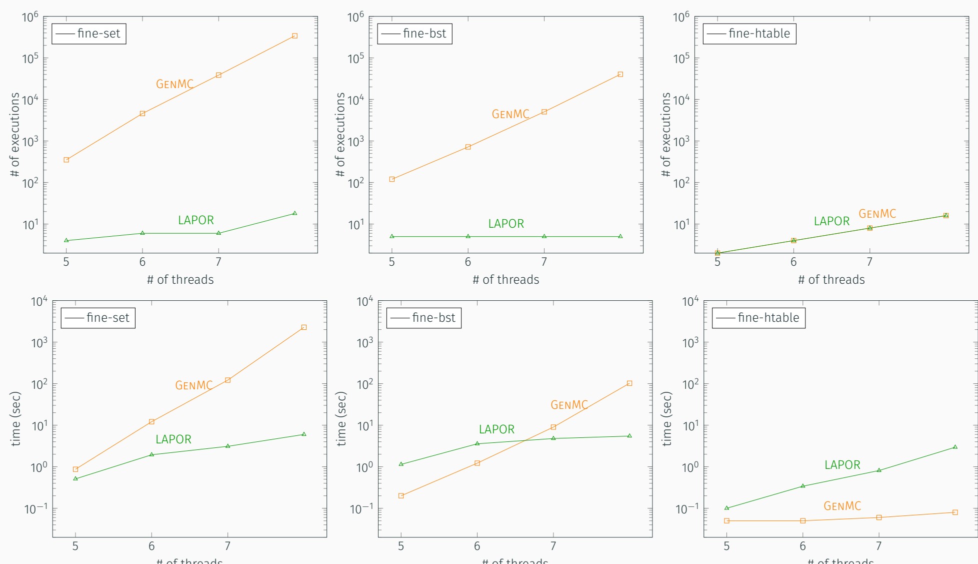
Fine-Grained Data-Structures



Fine-Grained Data-Structures



Fine-Grained Data-Structures



More in the paper

- Detailed description of the algorithm
- Formalization of the induced ordering's calculation
- More benchmarks and evaluation

Conclusions

Summary

- A Lock-Aware Partial Order Reduction Algorithm
  - independence at both instruction and critical section levels
  - sound, complete, and optimal
  - works for a variety of memory models
- LAPOR can be exponentially faster in programs with locks
- LAPOR is available at `github.com/MPI-SWS/genmc`

Conclusions

Summary

- A Lock-Aware Partial Order Reduction Algorithm
  - independence at both instruction and critical section levels
  - sound, complete, and optimal
  - works for a variety of memory models
- LAPOR can be exponentially faster in programs with locks
- LAPOR is available at `github.com/MPI-SWS/genmc`

Future work

- Extend the same idea to transactions and other synchronization idioms

Thank You!

Backup Slides

Treating Critical Sections As Atomic Blocks

$$[l = x = 0]$$

$$\begin{array}{l} lock(l); \\ x := 1; \\ x := 2; \\ unlock(l) \end{array} \;\Bigg|\Bigg|\; a := x$$