MUNI
FI

Extending the concurrency safety capacity of Slowbeast

- Mgr. Thesis defence - Suyash Shandilya
- Supervisor: prof. RNDr. Jan Strejček, Ph.D.

Symbolic Execution

- Takes a program as an input – C.
- Interpret program inputs as symbols
- Specific Program verification task – Data Race
- Slowbeast
  - Written in Python
  - Converts C to LLVM before analysis.

Path explosion with Concurrency

- Both threads currently active

global initialize: G

THREAD 1

a ← 1
b ← b + a
store **b** in G

THREAD 2

load G in c
d ← c - 3
store **d** in G

Path explosion with Concurrency

- Both threads currently active
- Possible interleavings:

global initialize: G

THREAD 1	THREAD 2
a ← 1	load G in c
b ← b + a	d ← c - 3
store **b** in G	store **d** in G

1: a ← 1
2: load G in c
2: d ← c - 3
2: store **d** in G
1: b ← b + a
1: store **b** in G

2: load G in c
1: a ← 1
2: d ← c - 3
2: store **d** in G
1: b ← b + a
1: store **b** in G

...

1: a ← 1
1: b ← b + a
2: load G in c
1: store **b** in G
2: d ← c - 3
2: store **d** in G

1: a ← 1
2: load G in c
1: b ← b + a
1: store **b** in G
2: d ← c - 3
2: store **d** in G

2: load G in c
1: a ← 1
1: b ← b + a
2: d ← c - 3
1: store **b** in G
2: store **d** in G

MUNI FI

Path explosion with Concurrency

- Both threads currently active
- Possible interleavings:

global initialize: G

THREAD 1

a ← 1
b ← b + a
store **b** in G

THREAD 2

load G in c
d ← c - 3
store **d** in G

1: a ← 1
2: load G in c
2: d ← c - 3
2: store **d** in G
1: b ← b + a
1: store **b** in G

2: load G in c
1: a ← 1
2: d ← c - 3
2: store **d** in G
1: b ← b + a
1: store **b** in G

...

1: a ← 1
1: b ← b + a
2: load G in c
1: store **b** in G
2: d ← c - 3
2: store **d** in G

1: a ← 1
2: load G in c
1: b ← b + a
1: store **b** in G
2: d ← c - 3
2: store **d** in G

2: load G in c
1: a ← 1
1: b ← b + a
2: d ← c – 3
1: store **b** in G
2: store **d** in G

Data Race Definition

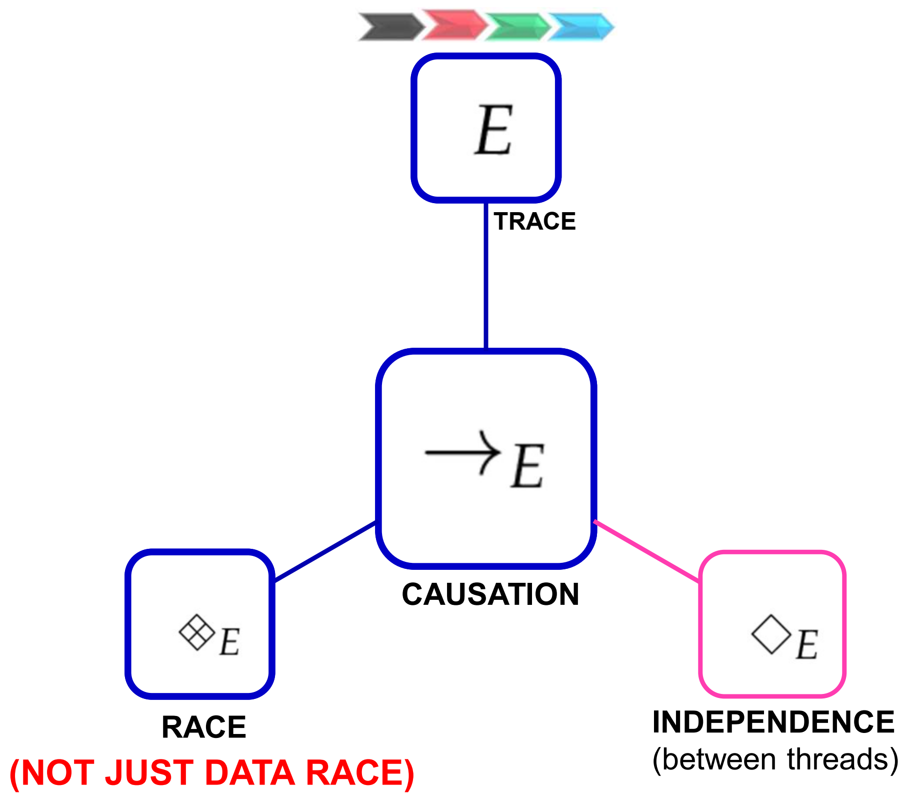
- Multiple threads access same memory location with at least one of the threads trying to write to the location.
- Value being written may be same or different.

Partial Order Reduction

- Thread agnostic exploration is often infeasible
- Source set – Dynamic Partial Order Reduction **[SDPOR]**
  - Optimal Dynamic Partial Order Reduction – Parosh Abdulla, et al. – Uppsala University, Sweden
  - ACM – Symposium on Principles of Programming Languages – San Diego, USA – Jan’14
- Independence relation between threads at certain program locations
- Analyzes execution sequence

MUNI
FI

Relations required



What is a race?

global initialize: G
global mutex lock: L

THREAD 1	THREAD 2
acq_lock L	acq_lock L
store 5 in G	store 6 in G
rel_lock L	rel_lock L

- Defined for a sequence
- Instructions whose order of occurrence if inverted, can lead to new states.
- **Lock Race**
- Only between successful acquisition of locks
- Data Race?

```
1: acq_lock L
2: acq_lock L
1: store 5 in G
1: rel_lock L
2: acq_lock L
2: store 6 in G
2: rel_lock L
```

. . .

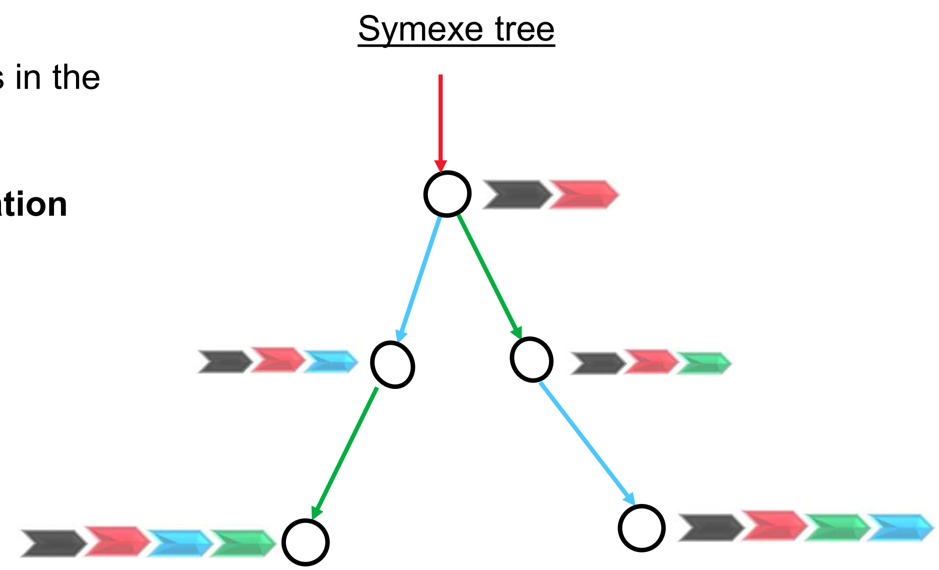
```
2: acq_lock L
1: acq_lock L
2: store 6 in G
2: rel_lock L
1: acq_lock L
1: store 5 in G
1: rel_lock L
```

```
2: acq_lock L
2: store 6 in G
1: acq_lock L
2: rel_lock L
1: acq_lock L
1: store 5 in G
1: rel_lock L
```

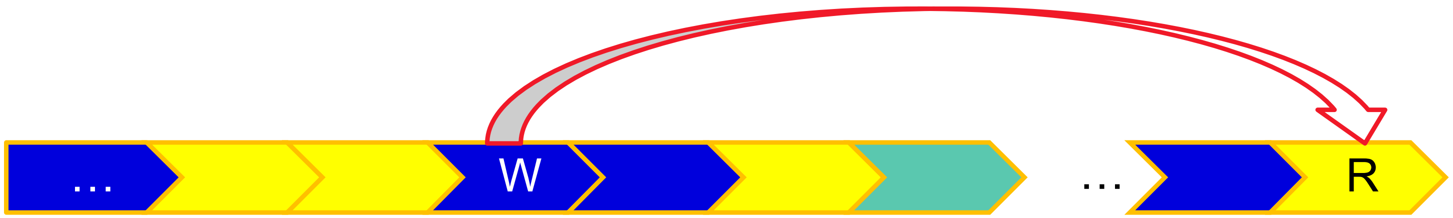
. . .

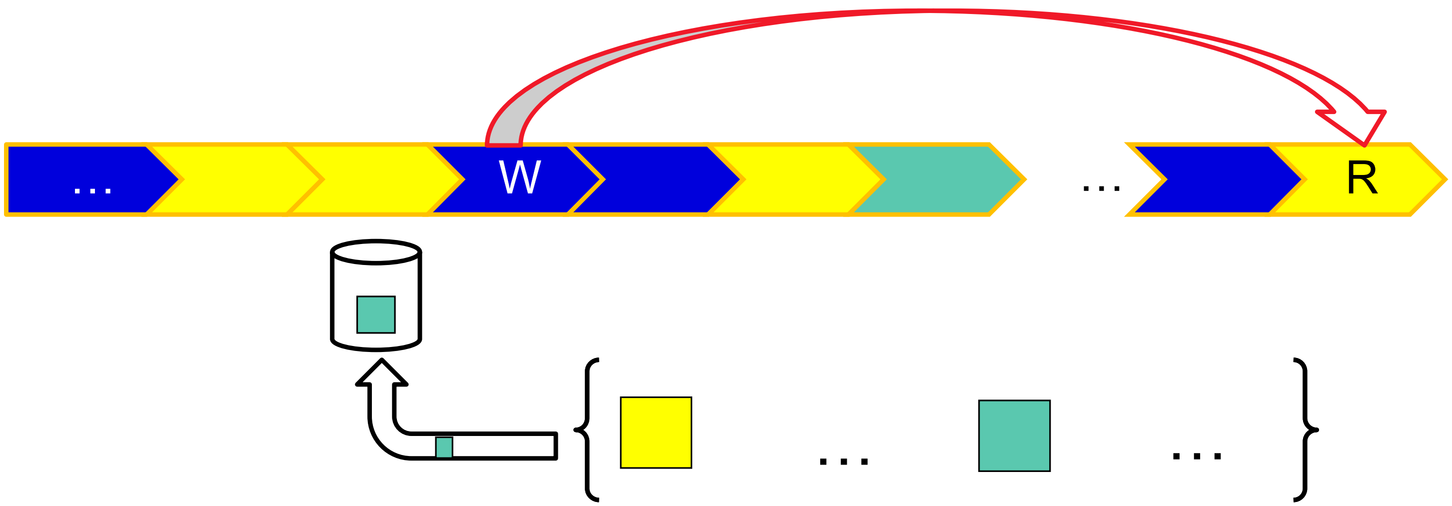
Using SDPOR with Symbolic execution

- Execute State then analyze trace
- Use the POR algorithm to
  1. Draws **causation relation** between the instructions in the sequence
  2. Use it to calculate **race** between instructions
  3. Updates the **Backtrack set** in the **forward exploration** (next slide)
  4. Updates the **Sleep set** when **backtracking**
- Data race detection?

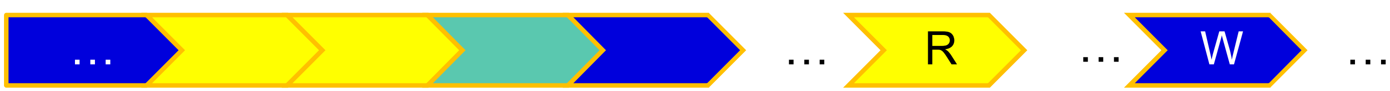


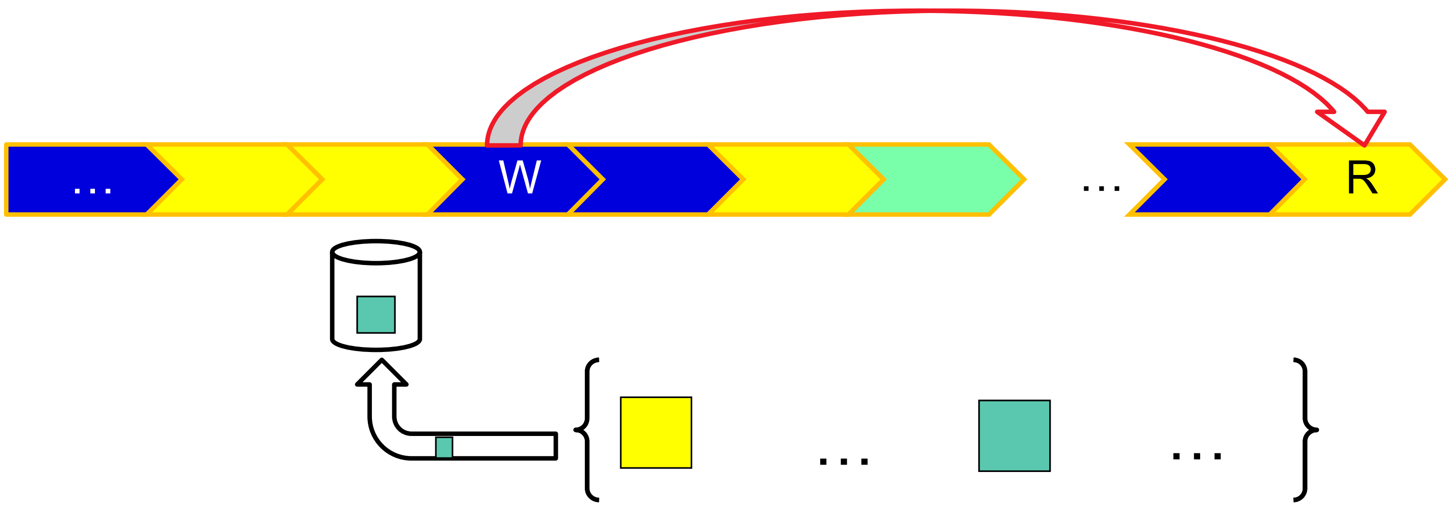
MUNI FI


...
W
...
R

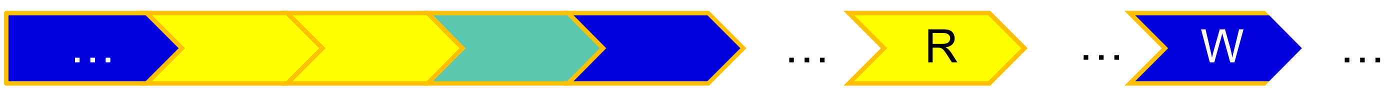


Subsequently…



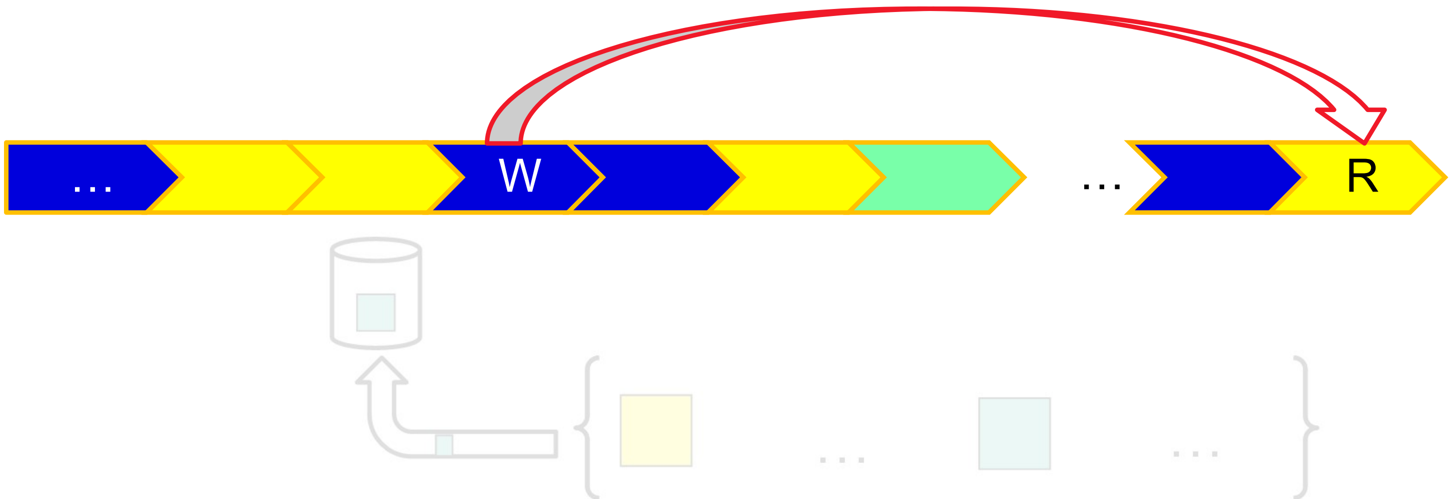


Subsequently…



But data race is implied by:

Data Race found!



Subsequently…

But data race is implied by:

Data Race found!

Previous work

- Implementation in Concuerror - Erlang
- Actor model of concurrency. – message passing
- Simpler causal relation
- No Mutex locks
- No data race

MUNI
FI

Impact of current work

- 22 files changed. 575 insertions (excluding tests)
- Limited concurrency support previously
- Existing bugs
- Other concurrency benchmarks
- Reusable causality relation

MUNI
FI

Experimental Evaluation

- SV-COMP – ConcurrencySafety - NoDataRace
- 1013 benchmarks in C
  - 792 do not have data race
  - 221 have data race
- benchexec for measuring resource utilization
- Maximum memory: 15GB. Maximum time: 15 minutes
- Top performing tools in the category in 2024 : Goblint and Ultimate Gemcutter
- Tools for comparison not run locally

Results

	Slowbeast	Goblint	Ultimate Gemcutter
Correct True	35	669	536
Correct False	26	0	144
Total Correct	**61**	**669**	**680**
Incorrect True	2	0	0
Incorrect False	0	0	1
Total Incorrect	**2**	**0**	**1**
Total Unknown	**951**	**344**	**332**

Unsupported benchmarks and incorrect results

- Atomic instructions and Instrumentations
- Other pthread_api
- string functions
- PTHREAD_MUTEX_INITIALIZER
- Incorrect results due to double pointers

Refined Results

	Slowbeast	Goblint	Ultimate Gemcutter
Correct True	35	160	104
Correct False	26	0	0
Total Correct	**61**	**160**	**104**
Incorrect True	2	0	0
Incorrect False	0	0	0
Total Incorrect	**2**	**0**	**0**
Timeout	2	1	47
Recursion Errors	122	N/A	N/A
Total Unknowns	**261**	**164**	**220**

Future work

- Extending to atomic instructions
- Debugging pointer resolution issues
- Recursive to iterative
- Integrating with Symbiotic

Thank you!

- Recursive
- Small and simple.
- Correct.
- No proof provided.
- Dos and **Don'ts list**

Algorithm 1: Source-DPOR algorithm

Initially $Explore(\langle\rangle, \emptyset)$;

$Explore(E, Sleep)$;

  if $\exists p \in (enabled(s_{[E]}) \setminus Sleep)$ **then**

    $backtrack(E) := \{p\}$;

    while $\exists p \in (backtrack(E) \setminus Sleep)$ **do**

      foreach $e \in dom(E)$ **such that** $(e \precsim_{E.p} next_{[E]}(p))$ **do**

        let $E' = pre(E, e)$;

        let $v = notdep(e, E).p$;

        if $I_{[E']}(v) \cap backtrack(E') = \emptyset$ **then**

          add some $q' \in I_{[E']}(v)$ to $backtrack(E')$;

      let $Sleep' := \{q \in Sleep \mid E \models p \Diamond q\}$;

      $Explore(E.p, Sleep')$;

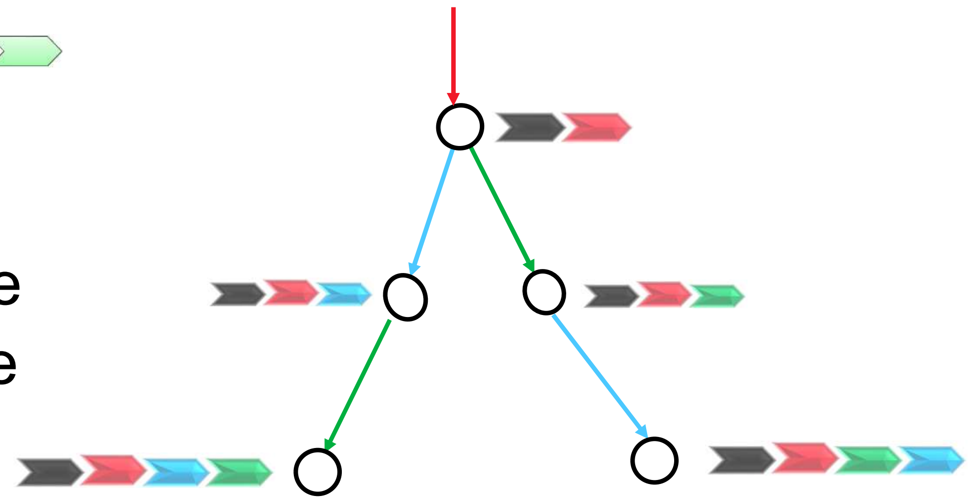
      add p to $Sleep$;

MUNI
FI

Traces vs states

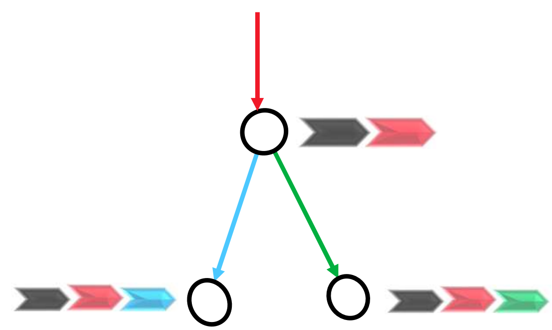
- Execution sequence = trace
- States ○
- Only traces
- Update trace then execute state
- Execute state then update trace

Symexe tree



Traces vs states

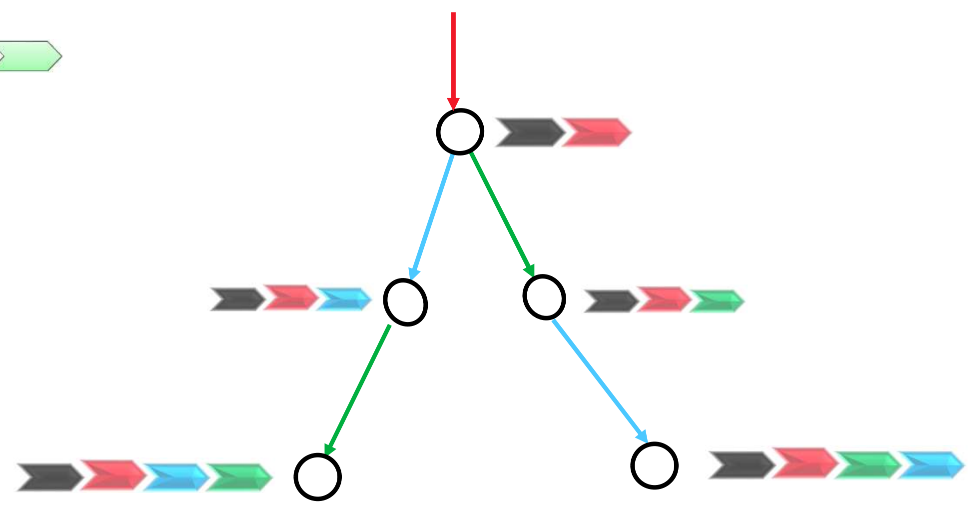
- Execution sequence = trace
- States
- Only traces


Symexe tree

Traces vs states

- Execution sequence = trace
- States
- Only traces

Symexe tree



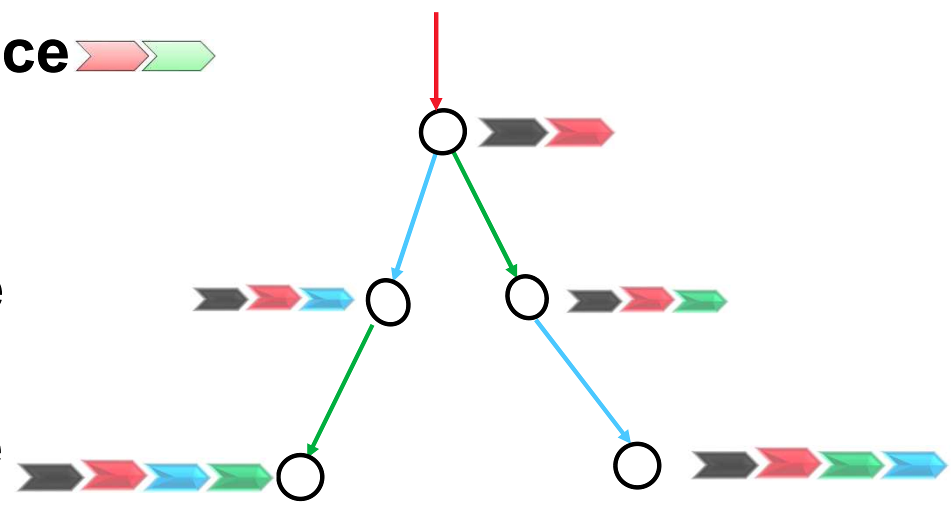
Traces vs states

- POR: Execution sequence = **trace**
- Symexe: States
- Atomic instructions
- Update trace then execute state

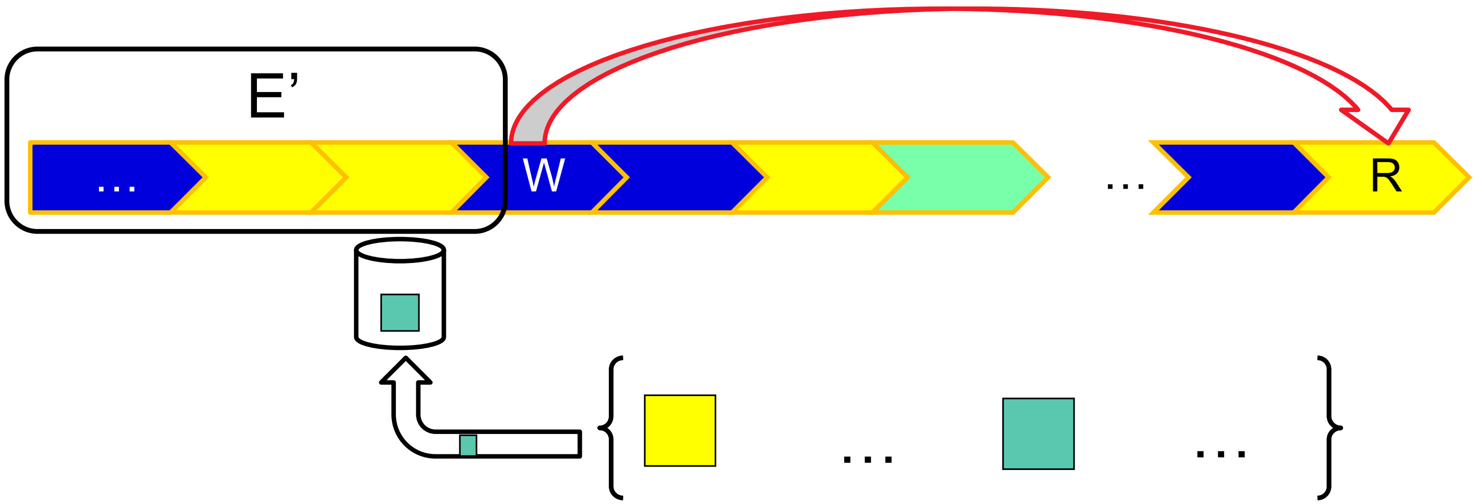
  OR

  Execute state then update trace

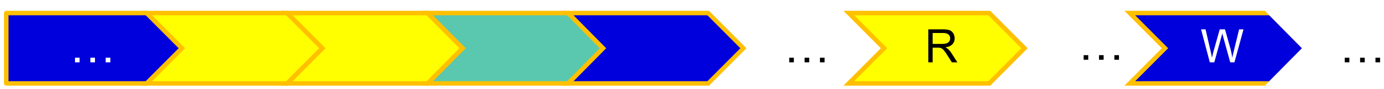
Symexe tree



MUNI FI

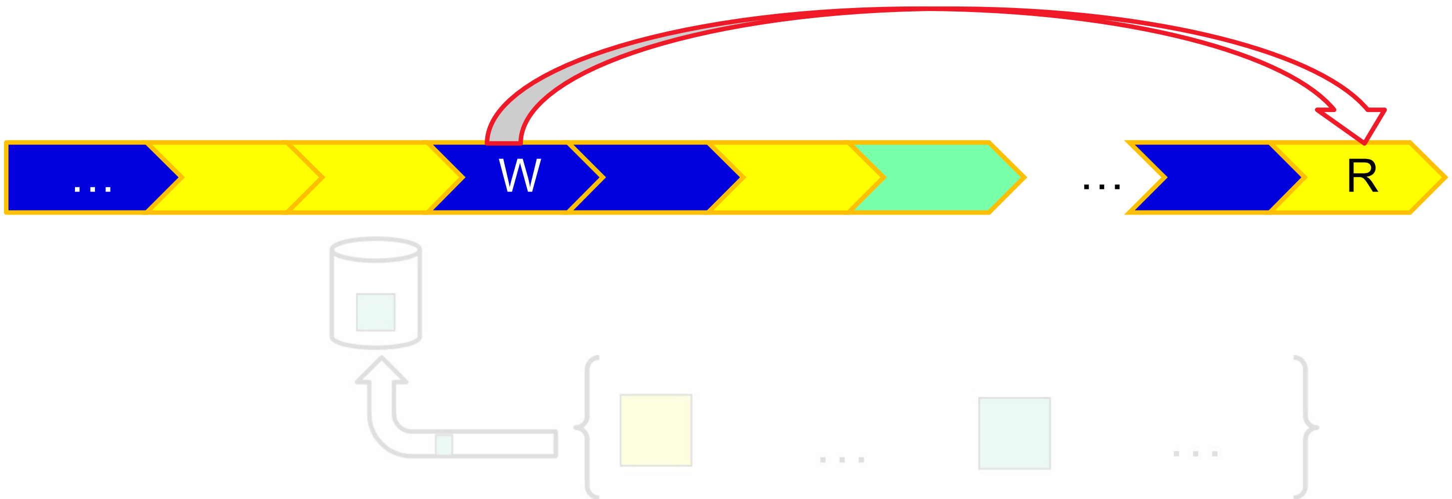


Subsequently…



Does it assure?

YES!



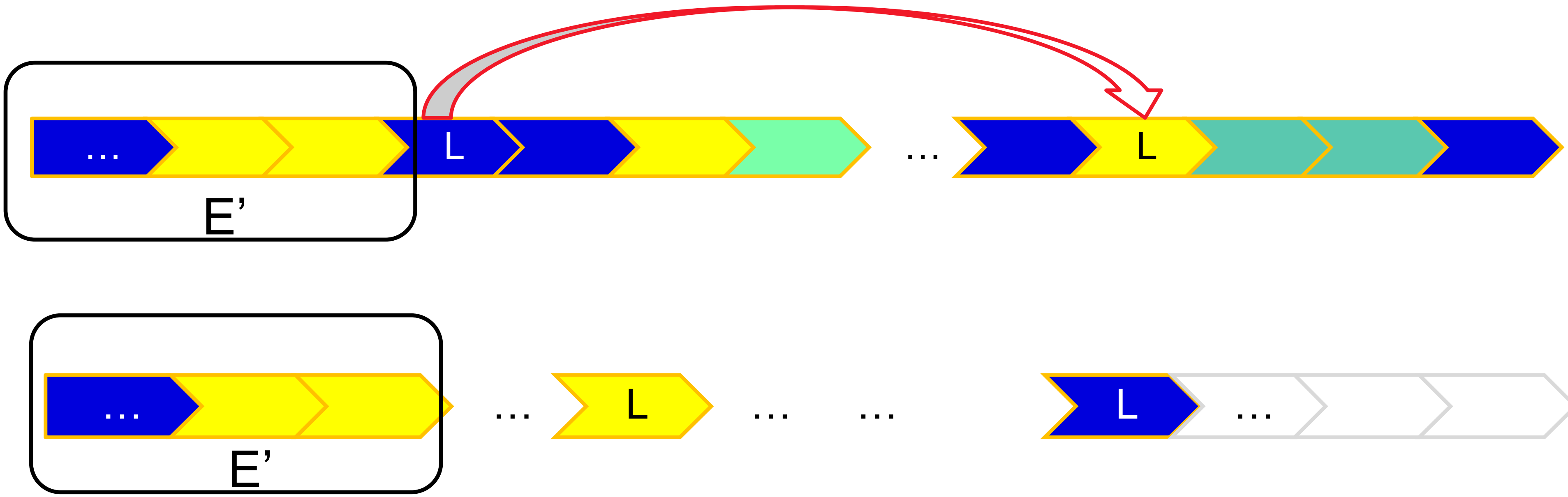
Subsequently…

Does it assure?

YES!

MUNI FI

LOCK RACE



Algorithm 1: Source-DPOR algorithm

Initially $Explore(\langle\rangle, \emptyset)$;

$Explore(E, Sleep)$;

  if $\exists p \in (enabled(s_{[E]}) \setminus Sleep)$ **then**

    $backtrack(E) := \{p\}$;

    while $\exists p \in (backtrack(E) \setminus Sleep)$ **do**

      foreach $e \in dom(E)$ **such that** $(e \precsim_{E.p} next_{[E]}(p))$ **do**

        let $E' = pre(E, e)$;

        let $v = notdep(e, E).p$;

        if $I_{[E']}(v) \cap backtrack(E') = \emptyset$ **then**

          add some $q' \in I_{[E']}(v)$ to $backtrack(E')$;

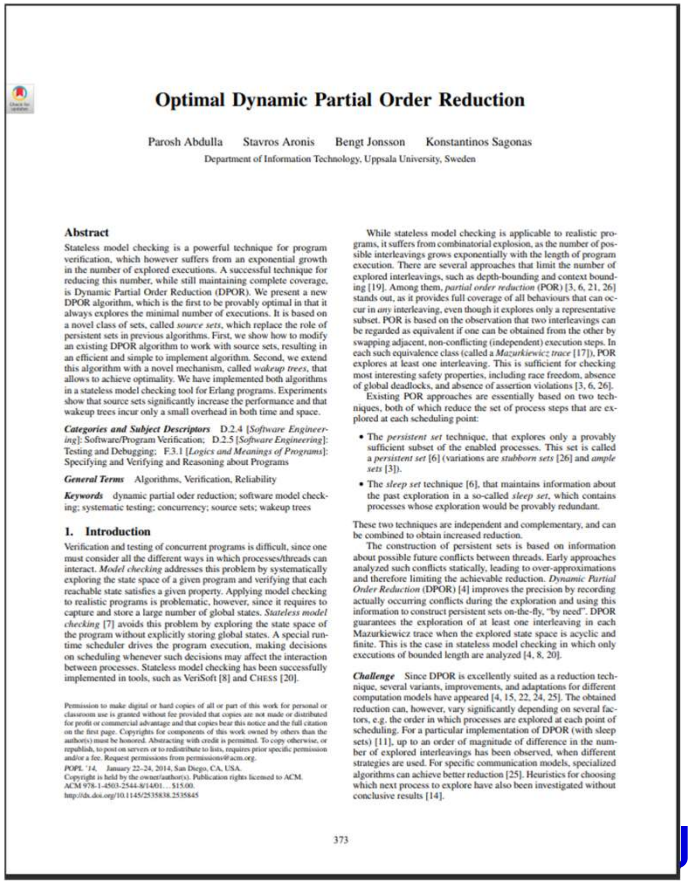
      let $Sleep' := \{q \in Sleep \mid E \models p \diamond q\}$;

      $Explore(E.p, Sleep')$;

      add p to $Sleep$;

Original Work

- ODPOR vs SDPOR
- 1.5 pages in the original paper
- Cite the paper properly.



Optimal Dynamic Partial Order Reduction

Parosh Abdulla Stavros Aronis Bengt Jonsson Konstantinos Sagonas
Department of Information Technology, Uppsala University, Sweden

Abstract

Stateless model checking is a powerful technique for program verification, which however suffers from an exponential growth in the number of explored executions. A successful technique for reducing this number, while still maintaining complete coverage, is Dynamic Partial Order Reduction (DPOR). We present a new DPOR algorithm, which is the first to be provably optimal in that it always explores the minimal number of executions. It is based on a novel class of sets, called *source sets*, which replace the role of persistent sets in previous algorithms. First, we show how to modify an existing DPOR algorithm to work with source sets, resulting in an efficient and simple to implement algorithm. Second, we extend this algorithm with a novel mechanism, called *wakeup trees*, that allows to achieve optimality. We have implemented both algorithms in a stateless model checking tool for Erlang programs. Experiments show that source sets significantly increase the performance and that wakeup trees incur only a small overhead in both time and space.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Verification, Reliability

Keywords dynamic partial oder reduction; software model checking; systematic testing; concurrency; source sets; wakeup trees

1. Introduction

Verification and testing of concurrent programs is difficult, since one must consider all the different ways in which processes/threads can interact. *Model checking* addresses this problem by systematically exploring the state space of a given program and verifying that each reachable state satisfies a given property. Applying model checking to realistic programs is problematic, however, since it requires to capture and store a large number of global states. *Stateless model checking* [7] avoids this problem by exploring the state space of the program without explicitly storing global states. A special run-time scheduler drives the program execution, making decisions on scheduling whenever such decisions may affect the interaction between processes. Stateless model checking has been successfully implemented in tools, such as VeriSoft [8] and CHESS [20].

POPL '14, January 22–24, 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01…$15.00.
http://dx.doi.org/10.1145/2535838.2535845

While stateless model checking is applicable to realistic programs, it suffers from combinatorial explosion, as the number of possible interleavings grows exponentially with the length of program execution. There are several approaches that limit the number of explored interleavings, such as depth-bounding and context bounding [19]. Among them, *partial order reduction* (POR) [3, 6, 21, 26] stands out, as it provides full coverage of all behaviours that can occur in *any* interleaving, even though it explores only a representative subset. POR is based on the observation that two interleavings can be regarded as equivalent if one can be obtained from the other by swapping adjacent, non-conflicting (independent) execution steps. In each such equivalence class (called a *Mazurkiewicz trace* [17]), POR explores at least one interleaving. This is sufficient for checking most interesting safety properties, including race freedom, absence of global deadlocks, and absence of assertion violations [3, 6, 26].

Existing POR approaches are essentially based on two techniques, both of which reduce the set of process steps that are explored at each scheduling point:

- The *persistent set* technique, that explores only a provably sufficient subset of the enabled processes. This set is called a *persistent set* [6] (variations are *stubborn sets* [26] and *ample sets* [3]).
- The *sleep set* technique [6], that maintains information about the past exploration in a so-called *sleep set*, which contains processes whose exploration would be provably redundant.

These two techniques are independent and complementary, and can be combined to obtain increased reduction.

The construction of persistent sets is based on information about possible future conflicts between threads. Early approaches analyzed such conflicts statically, leading to over-approximations and therefore limiting the achievable reduction. *Dynamic Partial Order Reduction* (DPOR) [4] improves the precision by recording actually occurring conflicts during the exploration and using this information to construct persistent sets on-the-fly, "by need". DPOR guarantees the exploration of at least one interleaving in each Mazurkiewicz trace when the explored state space is acyclic and finite. This is the case in stateless model checking in which only executions of bounded length are analyzed [4, 8, 20].

Challenge Since DPOR is excellently suited as a reduction technique, several variants, improvements, and adaptations for different computation models have appeared [4, 15, 22, 24, 25]. The obtained reduction can, however, vary significantly depending on several factors, e.g. the order in which processes are explored at each point of scheduling. For a particular implementation of DPOR (with sleep sets) [11], up to an order of magnitude of difference in the number of explored interleavings has been observed, when different strategies are used. For specific communication models, specialized algorithms can achieve better reduction [25]. Heuristics for choosing which next process to explore have also been investigated without conclusive results [14].

Results in thesis

- Benchexec
- External tools not run locally
- SV-COMP source

	Slowbeast	Goblint	UGemCutter
Correct True	35	669	536
Correct False	26	0	144
Total Correct	**61**	**669**	**680**
Incorrect True	2	0	0
Incorrect False	0	0	1
Total Incorrect	**2**	**0**	**1**
Unknown	931	344	332
Score	32	1338	1200

Errors

dryes > program.ll

```
fun thread1(a3)
  arg = alloc 4:32b bytes
  store (*)a3 to (*)arg
   ; llvm :   store i8 49, i8* @v, align 1, !dbg !23
   ; dbgloc : ('/home/xshandil/ben_shandilya/slowbeast/dryes.c', 5, 5)
  store (8b)49:8b to (*)g1
   ; llvm :   ret i8* null, !dbg !24
```

C dryes.c > thread1(void *)

```c
#include <pthread.h>
char v;
void *thread1(void *arg)
{
  v = '1';
  return 0;
}
```

Errors

gobl-error1 > program.ll

```
fun t_fun(a4)
  arg = alloc 4:32b bytes
   ; llvm :   store i8* %0, i8** %2, align 4
  store (*)a4 to (*)arg
   ; llvm :   %3 = load i32*, i32** @s, align 4, !dbg !61
   ; dbgloc : ('gobl-error1.i', 990, 3)
   ; llvm :   %4 = getelementptr inbounds i32, i32* %3, i32 0, !dbg !61
   ; dbgloc : ('gobl-error1.i', 990, 3)
  x24: * = load (*)g2
   ; llvm :   store i32 8, i32* %4, align 4, !dbg !62
   ; dbgloc : ('gobl-error1.i', 990, 8)
  store (32b)8:32b to (*)x24
   ; llvm :   ret i8* null, !dbg !63
   ; dbgloc : ('gobl-error1.i', 991, 3)
  ret (*)ptr(0:32b, 0:32b)

nuf
```

gobl-error1.i > t_fun(void *)

```c
extern int getloadavg (double __loa
        __attribute__ ((__nothrow__ ,

int* s;
pthread_mutex_t mutex = { { 0, 0, 0
void *t_fun(void *arg) {
  s[0] = 8;
  return ((void *)0);
}
int main(void) {
  pthread_t id;
  s = (int*)malloc(sizeof(int));
  pthread_create(&id, ((void *)0),
  s[0] = 9;
  pthread_join (id, ((void *)0));
  return 0;
}
```

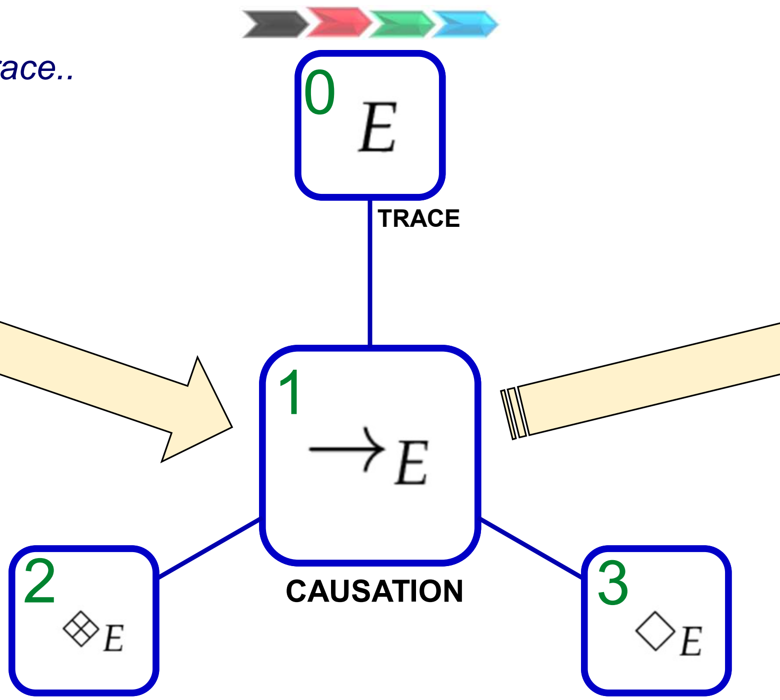
***Any** happens-before relation on a trace..*

- Must be a partial order on E

- Must order all instructions of a single thread.

- Must induce a set of equivalent traces with the same happens before relation.

...



RACE
(NOT DATA RACE)

- Relates **instructions**
- Different threads
- **Existence** of a **non-transitive** causation relation.
- ~~Commutable.~~ Reversible.

INDEPENDENCE

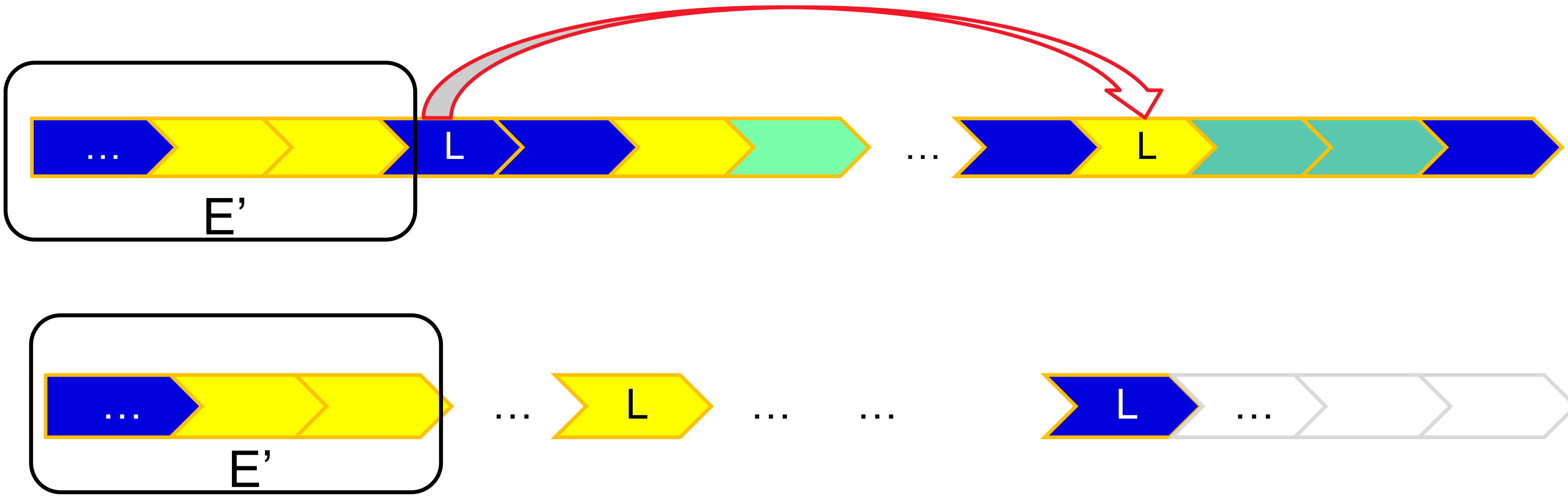
- Relates **threads**
- Absence of causation between ***next*** instructions of two threads.

*For **symbolic execution of LLVM program***

- Instruction writing from the same memory location must be related.
- Successful lock acquiring instructions to the same lock should be related.
- Unlock should relate to the instruction **after** the successful lock instruction by another thread.
- Create instruction must relate to the first instruction of the created thread**...**

M U N I
F I

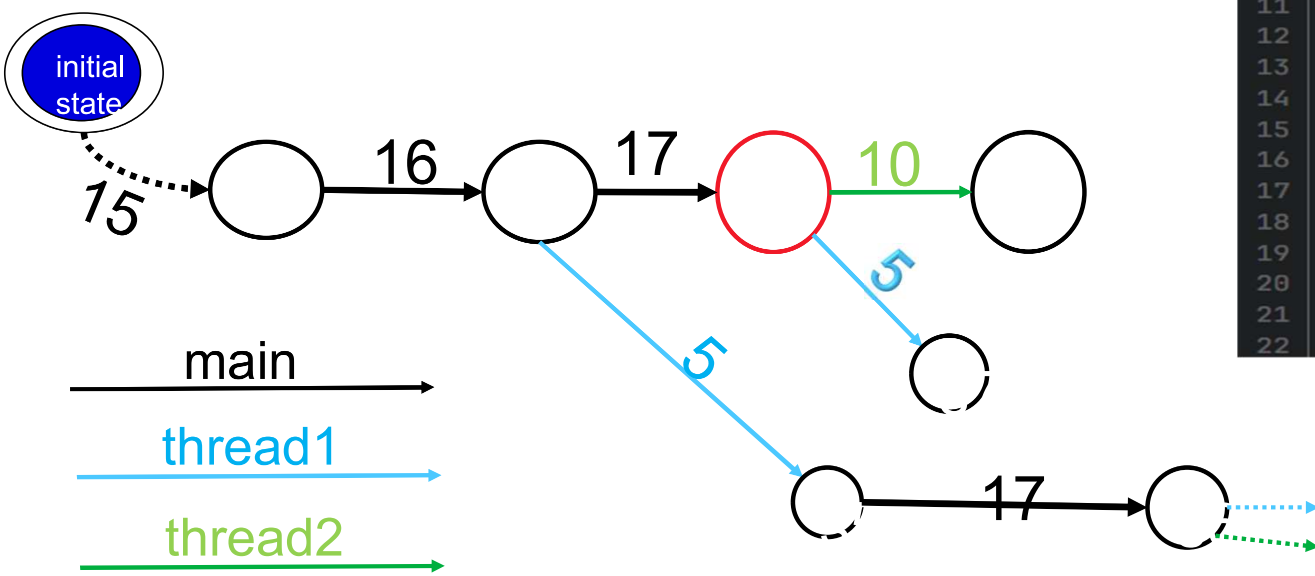
LOCK RACE



MUNI
FI

Program as an LTS

– Labelled Transition System

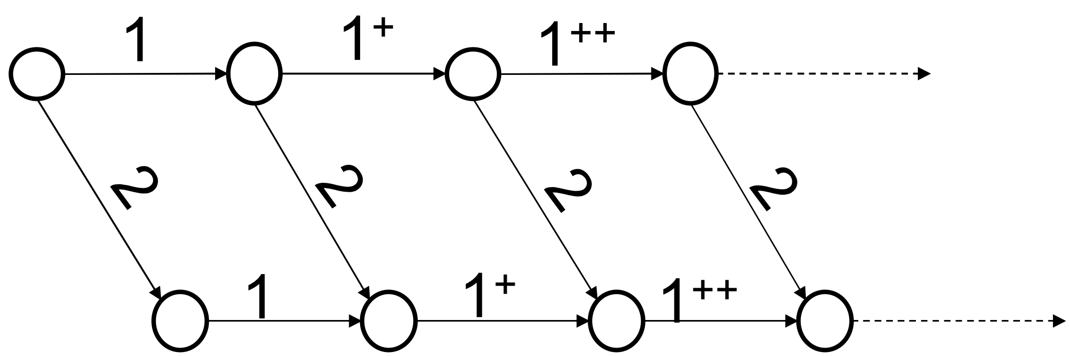



```c
#include <pthread.h>
char v;
void *thread1(void *arg)
{
  v = '1';
  return 0;
}
void *thread2(void *arg)
{
  v = '2';
  return 0;
}
int main()
{
  pthread_t t1, t2;
  pthread_create(&t1, 0, thread1, 0);
  pthread_create(&t2, 0, thread2, 0);
  pthread_join(t1, 0);
  pthread_join(t2, 0);

  return 0;
}
```

Partial Order Reduction

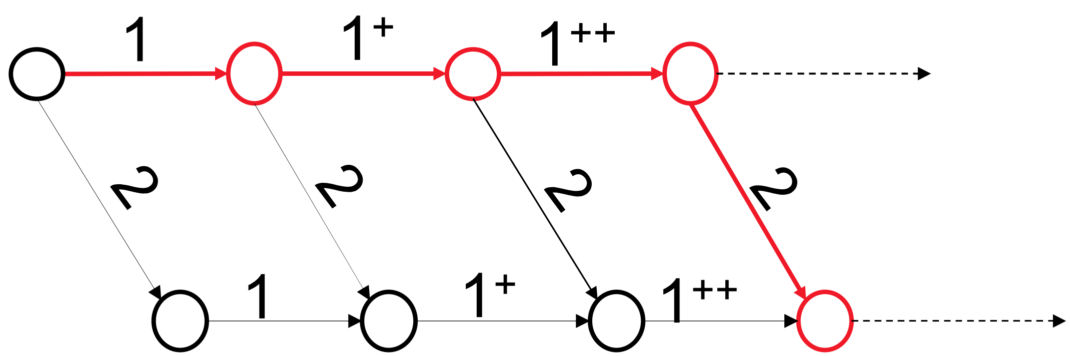
- `+` indicates next instruction
- 1 and 2 are instructions from two different threads.



MUNI
FI

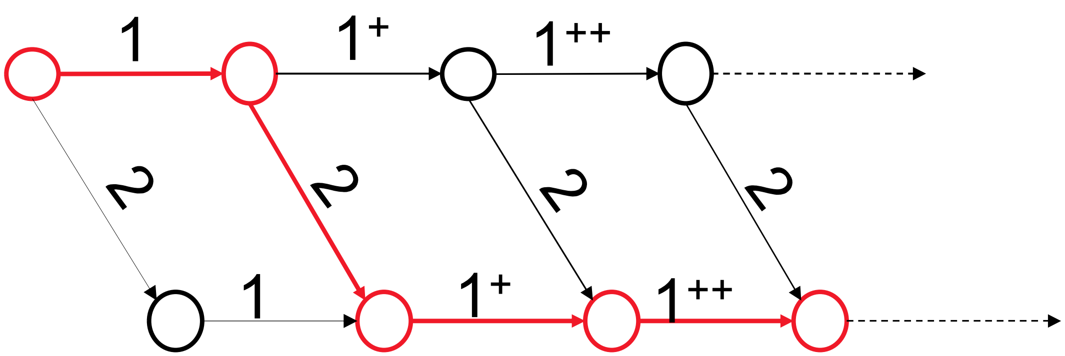
Partial Order Reduction

- **`+`** indicates next instruction
- **1** and **2** are instructions from two different threads.
- **If 1** and **2** are **independent,** all red paths are equivalent



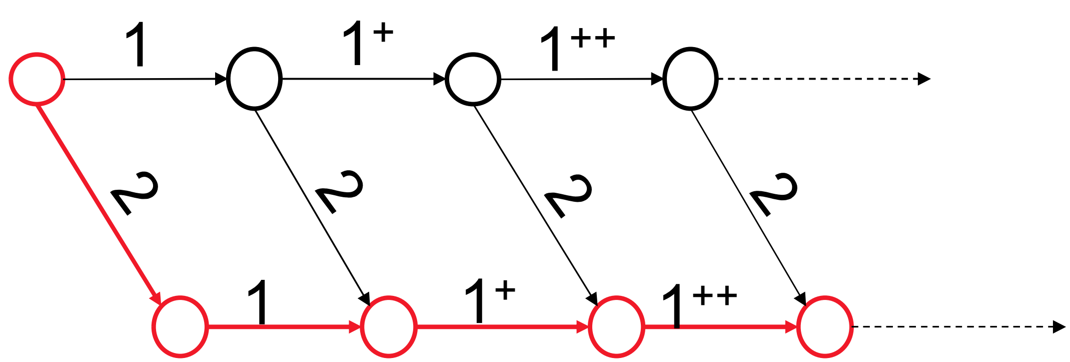
Partial Order Reduction

- **`+`** indicates next instruction
- **1** and **2** are instructions from two different threads.
- **If 1** and **2** are **independent,** all red paths are equivalent



Partial Order Reduction

- **`+`** indicates next instruction
- **1** and **2** are instructions from two different threads.
- **If 1** and **2** are **independent,** all red paths are equivalent



Algorithm 1: Source-DPOR algorithm

Initially $Explore(\langle\rangle, \emptyset)$;

$Explore(E, Sleep)$;

  if $\exists p \in (enabled(s_{[E]}) \setminus Sleep)$ **then**

    $backtrack(E) := \{p\}$;

    while $\exists p \in (backtrack(E) \setminus Sleep)$ **do**

      foreach $e \in dom(E)$ **such that** $(e \precsim_{E.p} next_{[E]}(p))$ **do**

        let $E' = pre(E, e)$;

        let $v = notdep(e, E).p$;

        if $I_{[E']}(v) \cap backtrack(E') = \emptyset$ **then**

          add some $q' \in I_{[E']}(v)$ to $backtrack(E')$;

      let $Sleep' := \{q \in Sleep \mid E \models p \Diamond q\}$;

      $Explore(E.p, Sleep')$;

      add p to $Sleep$;

Algorithm 1: Source-DPOR algorithm

Initially $Explore(\langle\rangle, \emptyset)$;

$Explore(E, Sleep)$;

if $\exists p \in (enabled(s_{[E]}) \setminus Sleep)$ **then**

$backtrack(E) := \{p\}$;

while $\exists p \in (backtrack(E) \setminus Sleep)$ **do**

foreach $e \in dom(E)$ **such that** $(e \precsim_{E.p} next_{[E]}(p))$ **do**

let $E' = pre(E, e)$;

let $v = notdep(e, E).p$;

if $I_{[E']}(v) \cap backtrack(E') = \emptyset$ **then**

add some $q' \in I_{[E']}(v)$ to $backtrack(E')$;

let $Sleep' := \{q \in Sleep \mid E \models p \Diamond q\}$;

$Explore(E.p, Sleep')$;

add p to $Sleep$;

Algorithm 1: Source-DPOR algorithm

Initially $Explore(\langle\rangle, \emptyset)$;

$Explore(E, Sleep)$;

  if $\exists p \in (enabled(s_{[E]}) \setminus Sleep)$ **then**

    $backtrack(E) := \{p\}$;

    while $\exists p \in (backtrack(E) \setminus Sleep)$ **do**

      foreach $e \in dom(E)$ **such that** $(e \precsim_{E.p} next_{[E]}(p))$ **do**

        let $E' = pre(E, e)$;

        let $v = notdep(e, E).p$;

        if $I_{[E']}(v) \cap backtrack(E') = \emptyset$ **then**

          add some $q' \in I_{[E']}(v)$ to $backtrack(E')$;

      let $Sleep' := \{q \in Sleep \mid E \models p \diamond q\}$;

      $Explore(E.p, Sleep')$;

      add p to $Sleep$;

MUNI FI