

# Program verification under weak memory consistency

Viktor Vafeiadis

MPI-SWS

Marktoberdorf, August 2019

Stateless model checking  
for weak memory models

# Stateless model checking (SMC)

## Stateless model checking

- ▶ Explore all thread interleavings
- ▶ *without* explicitly storing the visited states.

## Dynamic partial order reduction (DPOR)

[Flanagan and Godefroid 2005]

- ▶ Avoid exploring same interleavings.
- ▶ Source-DPOR, Optimal-DPOR [Abdulla et al. 2014]

## SMC for weak memory models

- ▶ Model buffers explicitly.
- ▶ Yet more “interleavings” to explore...

SMC for weak memory is **slower** than for SC.

	<i>Nidhugg</i> SC	<i>Nidhugg</i> TSO	<i>Nidhugg</i> PSO
linuxrwlocks(2)	0.22	0.25	0.33
linuxrwlocks(3)	37.65	43.41	65.14
ms-queue(2)	0.45	0.49	0.69
ms-queue(3)	21.21	23.13	36.12
qspinlock(2)	0.11	0.11	0.11
qspinlock(3)	15.78	17.12	26.62

SMC for weak memory is **slower** than for SC.

	<i>Nidhugg</i> SC	<i>Nidhugg</i> TSO	<i>Nidhugg</i> PSO	<i>RCMC</i>
linuxrwlocks(2)	0.22	0.25	0.33	<b>0.08</b>
linuxrwlocks(3)	37.65	43.41	65.14	<b>7.71</b>
ms-queue(2)	0.45	0.49	0.69	<b>0.13</b>
ms-queue(3)	21.21	23.13	36.12	<b>4.37</b>
qspinlock(2)	0.11	0.11	0.11	<b>0.06</b>
qspinlock(3)	15.78	17.12	26.62	<b>3.27</b>

But it need not be so!

A novel technique for verifying concurrent C/C++ programs that enumerates all their consistent **execution graphs**

- ▶ without generating inconsistent graphs;
- ▶ without generating the same graph twice;
- ▶ without recording the graphs already generated.

RCMC is proven:

- ▶ **sound** and **complete**; and
- ▶ **optimal** in the absence of RMWs and SC accesses.

RCMC target RC11 (repaired C11).

It outperforms the state-of-the-art model checkers on a variety of benchmarks.

# Key idea

## Example

$x := 1 \parallel a := x \parallel x := 2$

$[x = 0]$

$\downarrow$   
 $x := 1$

$\downarrow$   
 $a := x$

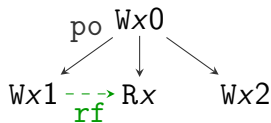
$\downarrow$   
 $x := 2$

## The standard approach

- ▶ Enumerate interleavings (total orders).
- ▶ Use an operational semantics.

## Our approach

- ▶ Enumerate **graphs** (partial orders).
- ▶ Use an **axiomatic semantics**.



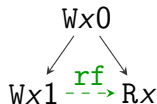
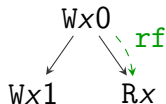
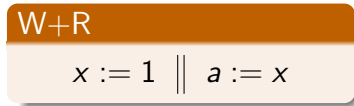
# Execution graphs in axiomatic memory models

## Events:

- ▶ writes  $Wxv$
- ▶ reads  $Rx$
- ▶ fences, etc.

## Relations:

- ▶ program order,  $po$
- ▶ reads-from,  $rf$
- ▶ other relations such as modification order,  $mo$



## + Consistency predicate

Here, we follow RC11 (repaired C11 with  $po \cup rf$  acyclic).

# Graph-based model checking

- Goal:** Enumerate all consistent execution graphs of  $P$  without
- ▶ generating the same graph multiple times; and
  - ▶ generating any inconsistent graphs.

For **stateless model checking**, also:

- ▶ avoid recording the set of graphs already generated.

Let's start with a *stateful* approach.

# Graph-based stateful model checking

Key property: consistency is prefix-closed

If an execution graph  $G$  is consistent, then every  $(po \cup rf)$ -prefix of  $G$  is also consistent.

## Naive approach:

- ▶ Record the set  $V$  of all graphs already generated.
- ▶ Initially,  $V$  contains only the empty execution graph.
- ▶ At each point, pick a graph  $G \in V$  and an event  $a$  such that  $G' = Add(G, a)$  is a consistent execution of  $P$ , and add  $G'$  to  $V$ .
- ▶ Repeat the previous until no new graphs can be added.

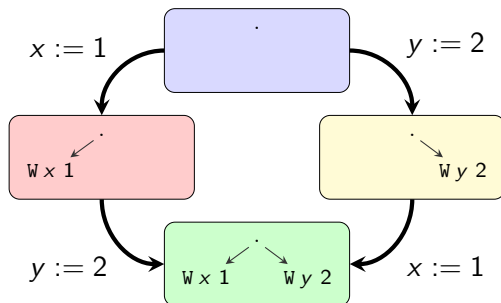
# The naive algorithm is too naive

## Observation

The order in which events are added is mostly irrelevant.

## Example

$x := 1 \parallel y := 2$



# Improving the naive algorithm

Fix an order in which events are added.

- ▶ e.g., in increasing thread ID order.

When adding a read  $r$ :

- ▶ Consider all possible writes that  $r$  could read from.

When adding a write  $w$ :

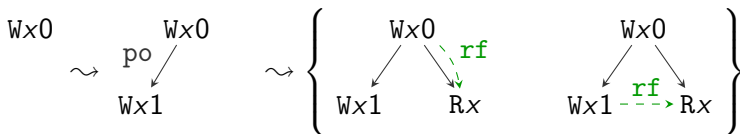
- ▶ Consider all possible placements of  $w$  in **mo** and also whether any existing reads can read from  $w$ .
- ▶ For any subset of such reads, “revisit” them:
  - ▶ Change their **rf**-incoming edges to read from  $w$ .
  - ▶ Delete any events  $(po \cup rf)^+$  after them.

# Simple example with the stateful algorithm

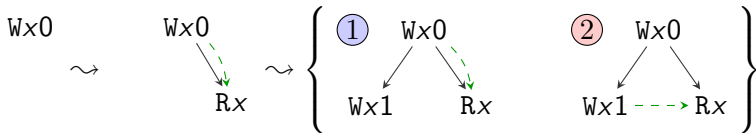
W+R

$x := 1 \parallel a := x$

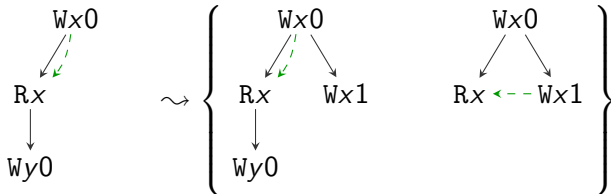
Add  $x := 1$  first:



Add  $a := x$  first:



# Revisiting removes successors

$$\begin{array}{l} a := x; \\ y := a \end{array} \parallel x := 1$$


# Graph-based *stateless* model checking

**Goal:** Enumerate all consistent execution graphs of  $P$  without

- ▶ generating the same graph multiple times;
- ▶ generating any inconsistent graphs; and
- ▶ recording the set of graphs already generated.

**Key challenge:**

- ▶ How to avoid repetition?

## Revisit sets

Record the set  $T$  of revisitable reads:

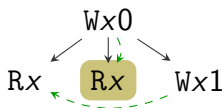
- ▶ *i.e.*, reads that **may be revisited** when extending  $G$ .

When adding a **read**  $r$ :

- ▶ Make  $r$  revisitable in only **one** of the subexecutions.

When adding a **write**  $w$  and revisiting a set  $R$  of reads:

- ▶ Carefully select which subsets of  $T$  will be revisited.
- ▶ Revisiting **removes** reads from  $T$ .

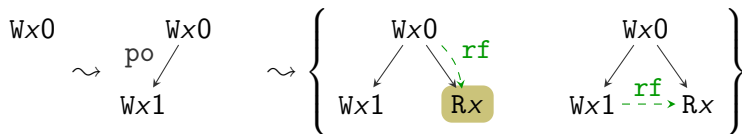


# Simple example

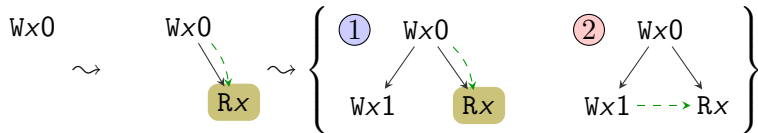
W+R

$x := 1 \parallel a := x$

Add  $x := 1$  first:



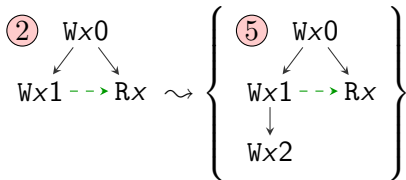
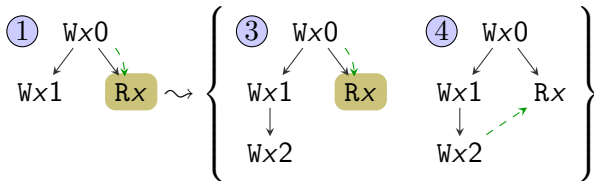
Add  $a := x$  first:



## A second example

COWW+R

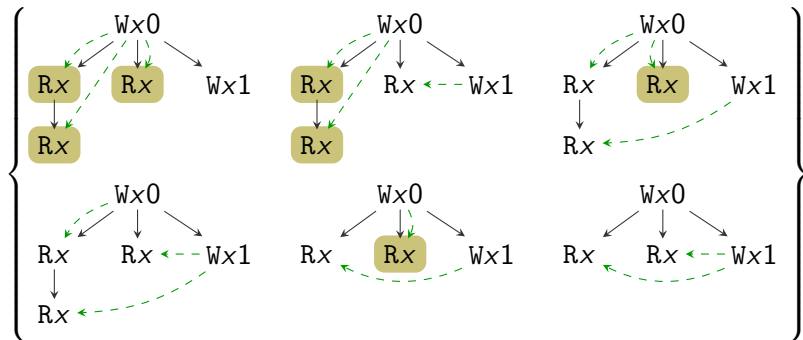
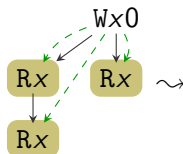
$x := 1;$  ||  $a := x$   
 $x := 2$  ||



# Yet another example

CO2RRW

$a := x;$  ||  $c := x$  ||  $x := 1$   
 $b := x$



- ▶ Reads reachable from a revisitable read are revisitable:

$$\text{codom}([T]; (\text{po} \cup \text{rf})^+; [\text{R}]) \subseteq T$$

- ▶ Every non-revisitable read in  $G$  is revisitable in some other visited execution.

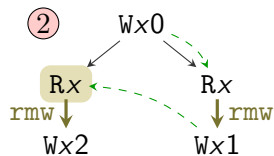
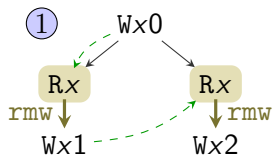
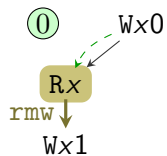
# Experiments with only reads and writes

	RCMC	CDS checker	Nidhugg SC	Nidhugg TSO	Nidhugg PSO	Nidhugg Power
readers(8)	<b>0.04</b>	0.05	0.26	0.29	0.34	193.70
readers(13)	<b>0.40</b>	2.14	7.25	8.15	11.18	<b>6.8 h</b>
readers(18)	<b>12.83</b>	128.91	350.48	422.10	444.34	<b>580 h</b>
lastzero0(5)	0.04	<b>0.01</b>	0.13	0.13	0.17	5.31
lastzero0(10)	<b>0.26</b>	0.78	13.21	14.80	19.28	1344.56
lastzero0(15)	<b>14.93</b>	50.28	<b>1.4 h</b>	<b>1.5 h</b>	<b>2 h</b>	<b>49 h</b>
lastzero1(5)	0.05	<b>0.02</b>	0.13	0.13	0.14	3.68
lastzero1(10)	<b>0.32</b>	7.03	2.29	2.74	3.42	954.91
lastzero1(15)	<b>15.02</b>	<b>0.8 h</b>	164.92	188.29	228.52	<b>34 h</b>
fibbench(3)	<b>0.12</b>	1.23	0.47	0.71	1.05	23.74
fibbench(4)	<b>1.17</b>	78.49	5.02	9.98	17.39	732.02
fibbench(5)	<b>19.42</b>	<b>1.4 h</b>	69.96	192.47	276.48	<b>5.7 h</b>

# Handling RMWs

FAIs

$\text{FAI}(x) \parallel \text{FAI}(x)$



# Experiments with RMWs

	<i>RCMC RCII</i>	<i>RCMC WRCII</i>	<i>CDS checker</i>	<i>Nidhugg SC</i>	<i>Nidhugg TSO</i>	<i>Nidhugg PSO</i>
indexer(12)	<b>0.05</b>		0.74	0.12	0.13	0.14
indexer(13)	<b>0.10</b>		111.29	0.29	0.31	0.38
indexer(14)	<b>0.56</b>		10 h	1.56	1.79	2.35
indexer(15)	<b>3.76</b>		*	12.74	14.55	18.05
casrot(6)	<b>0.05</b>		0.02	0.16	0.17	0.15
casrot(8)	<b>0.15</b>		0.24	1.10	1.21	1.55
casrot(10)	<b>2.13</b>		5.05	23.26	28.92	33.37
casw(4)	0.09	<b>0.05</b>	0.16	0.46	0.55	0.80
casw(5)	1.18	<b>0.14</b>	2.84	11.26	15.02	30.07
casw(6)	48.52	<b>1.26</b>	65.50	602.43	968.99	0.6 h
ainc(4)	<b>0.05</b>		0.24	0.11	0.11	0.11
ainc(5)	<b>0.05</b>		21.60	0.13	0.13	0.14
ainc(6)	<b>0.07</b>		1.3 h	0.27	0.29	0.35
binc(4)	<b>0.06</b>		100.97	0.21	0.23	0.29
binc(5)	<b>0.89</b>		*	3.90	4.60	6.30
binc(6)	<b>45.02</b>		*	167.68	196.83	242.40

## Summary

- ▶ **Sound** and **complete** SMC procedure for RC11.
- ▶ **Optimal** in the absence of RMWs and SC atomics.
- ▶ Works reasonably well even with RMWs and SC atomics.
- ▶ MC for WMM **does not** need to be harder than for SC!

## Extensions and other tools

- ▶ TRACER: For RA consistency [OOPSLA'18]
- ▶ GENMC: Parametric over the memory model [PLDI'19]
- ▶ SWSC: For sequential consistency [OOPSLA'19]
- ▶ LAPOR: Coarser equivalence for locks [OOPSLA'19]

<http://www.github.com/mpi-sws/genmc/>