Effective bounded verification of concurrent programs

Viktor Vafeiadis

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

1 June 2023

The grand challenge of software engineering

Produce **software** that is

- correct,
- efficient, and
- useful

with **minimal cost**

- in developer expertise and
- in developer time/effort.

The grand challenge of software engineering

Produce **software** that is

- correct, $\rightsquigarrow$ *use verification techniques*
- efficient, and $\rightsquigarrow$ *exploit parallelism*
- useful *application-dependent*

with **minimal cost**

- in developer expertise and
- in developer time/effort.

The grand challenge of software engineering

Produce **software** that is

- correct, ⇝ *use verification techniques*
- efficient, and ⇝ *exploit parallelism*
- useful *application-dependent*

with **minimal cost**

- in developer expertise and ⇝ *automated verification*
- in developer time/effort. ⇝ *with no false positives*

Software model checking (SMC)

Given a program P and a property Φ,
check that all executions of P satisfy Φ.

- It is **fully automated** ("push button" technique).
- Unsuccessful verification returns **error traces**,
  i.e. program traces that result in an error.

Software model checking (SMC)

Given a program P and a property Φ,
check that all executions of P satisfy Φ.

- It is **fully automated** ("push button" technique).
- Unsuccessful verification returns **error traces**,
  i.e. program traces that result in an error.
- It assumes programs are **bounded**.
- It is **slow** and it **does not scale well**.

The naive SMC approach does not scale!

There are ***way too many*** interleavings.

(exponential in the number of threads and the size of the program)

The naive SMC approach does not scale!

There are ***way too many*** interleavings.

(exponential in the number of threads and the size of the program)

But exploring all interleavings is ***unnecessary***.

- Many interleavings lead to the same outcome.

  DPOR: *Avoid exploring 'equivalent' interleavings*

- The same bug can be exposed by multiple interleavings.

  Bounding: *Explore only 'simple' interleavings*

The naive SMC approach does not scale!

There are ***way too many*** interleavings.

(exponential in the number of threads and the size of the program)

But exploring all interleavings is ***unnecessary***.

- Many interleavings lead to the same outcome.

  DPOR: *Avoid exploring 'equivalent' interleavings*

- The same bug can be exposed by multiple interleavings.

  Bounding: *Explore only 'simple' interleavings*

For best results, **combine** the two techniques.

Dynamic partial order reduction (DPOR)

- Two interleavings are **equivalent** if they agree on the order of racy accesses.

  e.g., $x := 1 \,;\, y := 1 \,;\, a := y$ *and* $y := 1 \,;\, x := 1 \,;\, a := y$

- Equivalent interleavings have the same outcome.

Correctness: Explore *at least one* interleaving per equiv. class

Optimality: Explore *exactly one* interleaving per equiv. class

TruSt: State-of-the-art in DPOR

- Correct, optimal, highly parallelizable;
- Works with almost any weak memory model;
- Has a small memory footprint (polynomial); and
- Has publicly available implementation (genmc).

Key ideas:

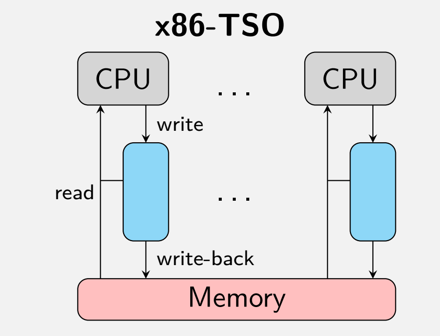
1. Represent equivalence classes as ***execution graphs***.
2. Generate all consistent graphs of P incrementally.
3. Constrain reversals via a maximality condition.

Execution graphs

Store buffering (SB)

Initially, $x = y = 0$.

$x := 1;$	$y := 1;$
$a := y$ // 0	$b := x$ // 0

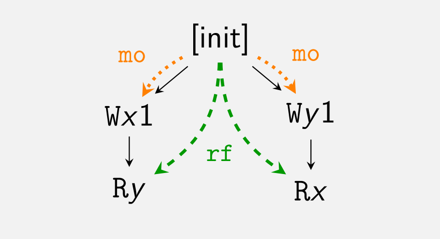


Execution graphs

Store buffering (SB)

Initially, $x = y = 0$.

$x := 1;$	$y := 1;$
$a := y$ *// 0*	$b := x$ *// 0*



program order (po), reads-from (rf), modification order (mo)

TruSt: Basic MC algorithm

Construct all consistent execution graphs incrementally

- Fix insertion order (e.g. increasing thread ID order)

TruSt: Basic MC algorithm

Construct all consistent execution graphs incrementally

- Fix insertion order (e.g. increasing thread ID order)

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

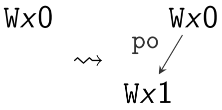
$\mathtt{W}x0$

TruSt: Basic MC algorithm

Construct all consistent execution graphs incrementally

- Fix insertion order (e.g. increasing thread ID order)

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



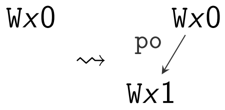
TruSt: Basic MC algorithm

Construct all consistent execution graphs incrementally

- Fix insertion order (e.g. increasing thread ID order)

Read r: Consider all possible writes that r could read from.

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



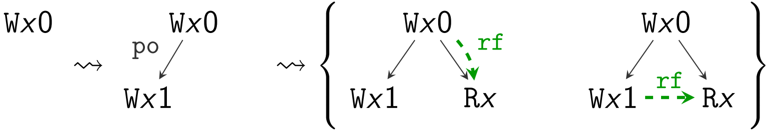
TruSt: Basic MC algorithm

Construct all consistent execution graphs incrementally

- Fix insertion order (e.g. increasing thread ID order)

Read r: Consider all possible writes that r could read from.

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



TruSt: Basic MC algorithm

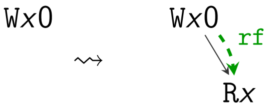
Construct all consistent execution graphs incrementally

- Fix insertion order (e.g. increasing thread ID order)

Read r: Consider all possible writes that r could read from.

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

Add $a := x$ first



TruSt: Basic MC algorithm

Construct all consistent execution graphs incrementally

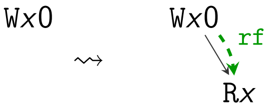
- Fix insertion order (e.g. increasing thread ID order)

Read r: Consider all possible writes that r could read from.

Write w: Revisit existing reads to instead read from w.

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

Add $a := x$ first



TruSt: Basic MC algorithm

Construct all consistent execution graphs incrementally

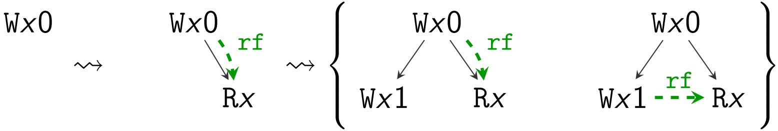
- Fix insertion order (e.g. increasing thread ID order)

Read r: Consider all possible writes that r could read from.

Write w: Revisit existing reads to instead read from w.

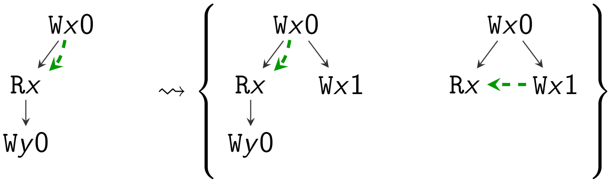
$$x := 1 \parallel a := x$$

Add $a := x$ first



TruSt: Revisits can delete events

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



Which events to remove on a $w \to r$ revisit?

(RCMC) those $(\mathrm{po} \cup \mathtt{rf})^+$-after r

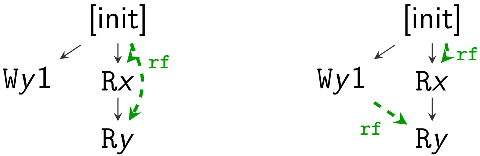
(GenMC) those added after r, not $(\mathrm{po} \cup \mathtt{rf})^+$-before w

Maximal graph extensions

Initially, $x = y = 0$.

$$y := 1 \;\Big\|\; \begin{array}{l} a := x \\ b := y \end{array} \;\Big\|\; x := 1$$

- The revisit of $a := x$ should happen in only one case:



- Choose the *maximal* one, where the revisited read and all events to be deleted were inserted maximally.

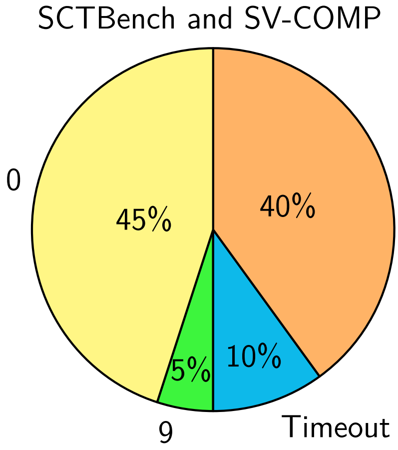
Preemption bounding

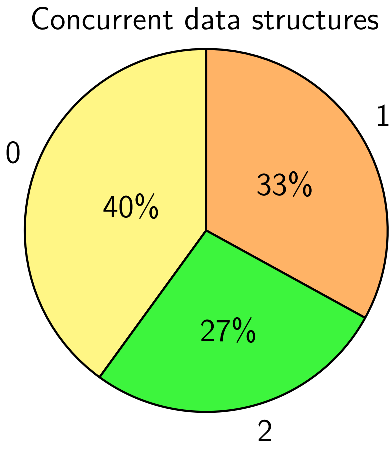
- Preemption: schedule a different thread although previous thread had not finished.
- Bugs tend to manifest with few thread preemptions.
  e.g., atomicity violations require only ***one*** *preemption*

$$
\begin{array}{l||l}
 & acquire(l) \\
a := x & \\
 & x := x + 1 \\
x := a + 1 & \\
 & release(l)
\end{array}
$$

- Explore only interleavings with up to $\boldsymbol{K}$ preemptions.
- #Interleavings is exponential in $\boldsymbol{K}$.

How many preemptions are needed to reveal bugs?

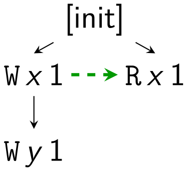




The combination is non-trivial (1/2)

Problem 1: POR-equivalent traces can have different number of preemptions.

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



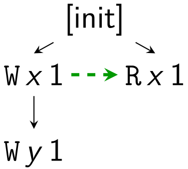
W x ; W y ; R x — *0 preemptions*

W x ; R x ; W y — *1 preemption*

The combination is non-trivial (1/2)

Problem 1: POR-equivalent traces can have different number of preemptions.

$$x := 1 \quad \| \quad a := x$$
$$y := 1$$



W x ; W y ; R x — *0 preemptions*

W x ; R x ; W y — *1 preemption*

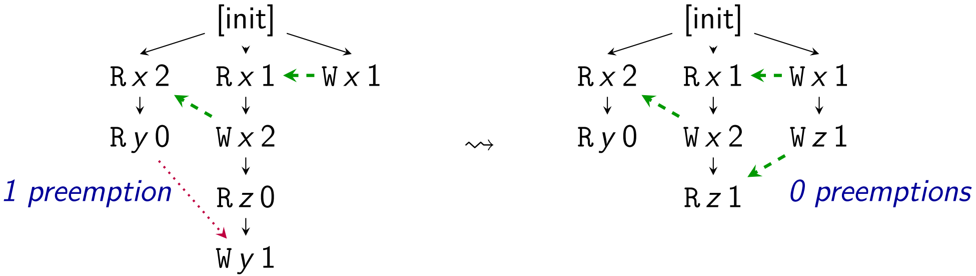
Solution: Define #preemptions of an execution graph:

- $\pi(G) \triangleq \min\{\pi(\tau) \mid \tau \text{ linearizes } G\}$.
- Calculating $\pi(G)$ is NP-complete.

The combination is non-trivial (2/2)

Problem 2: $\pi(.)$ is not monotone w.r.t. DPOR-visit order.

$a := x$	$c := x$	$x := 1$
$b := y$	$x := 2$	$z := 1$
	if $z = 0$ **then**	
	$\quad y := 1$	



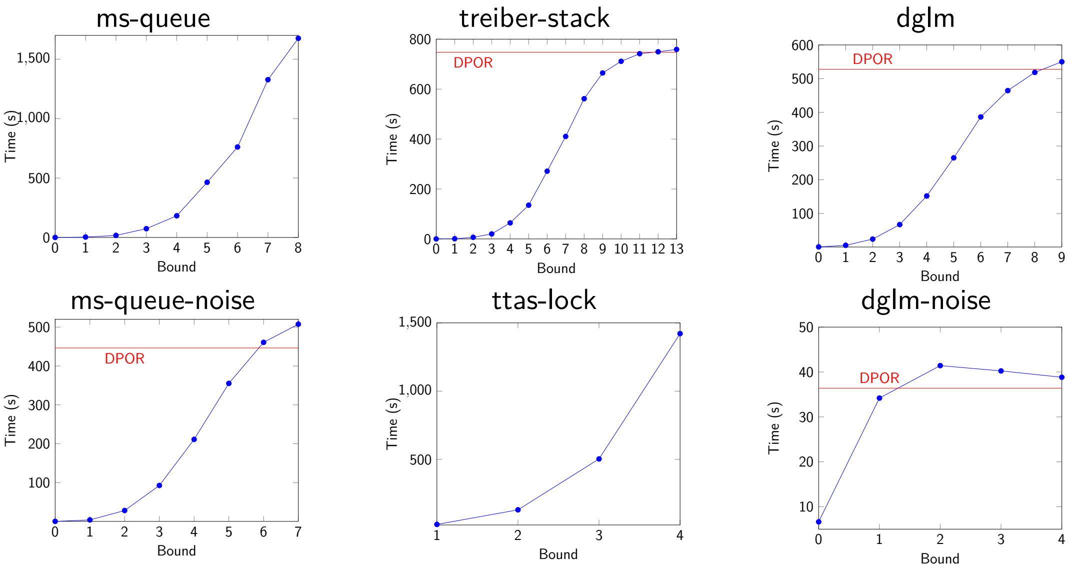
The combination is non-trivial (2/2)

Problem 2: $\pi(.)$ is not monotone w.r.t. DPOR-visit order.

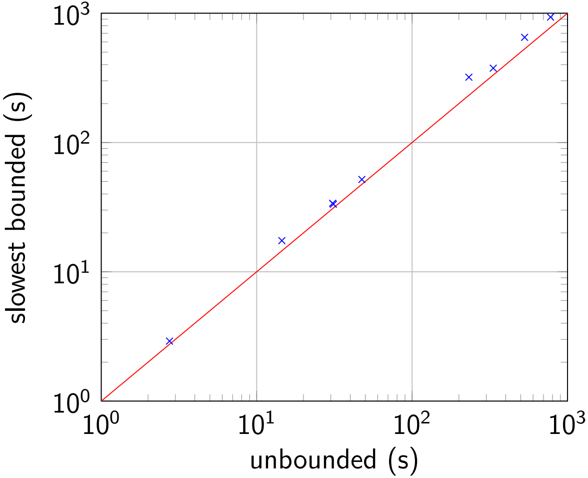
Solution: Allow some ***slack*** S

- Adapt TruSt to drop explorations with $\pi(G) > K + S$.
- Prove that $S = (\#\text{threads} - 2)$ is necessary.
- Prove that $S = (\#\text{threads} - 2)$ is sufficient.
- Optimal for 2 threads, gradually worsens with more threads: We may explore executions exceeding the bound, but never any two equivalent executions.

Up to what bound is it faster than plain DPOR?



Bound calculation overhead in CDs benchmarks



17% on average

A different communication bound?

Recap: Preemption bounding

- Calculating $\pi(G)$ is NP-complete.
- At every context switch, the scheduler can chose ***any*** other thread to run next.

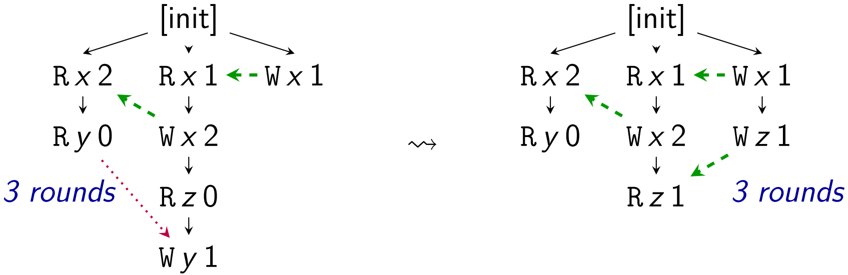
New idea: Restrict the scheduler's power

- Assume a round-robin scheduler.
- The only problem is detecting where the preemptions occur.
- Greedy approach $\rightsquigarrow$ linear in the size of the graph.

Bounding scheduling rounds

TruSt is **optimal** w.r.t. scheduling round bounding

- Execution extension is monotone w.r.t. scheduling rounds
- Revisiting a read does not decrease the scheduling rounds:
  The events removed by a revisit can be added in one round.



Comparing communication bounds

Preemptions

- Great for finding bugs
- Requires some ***slack*** on the exploration
- Bound checking can be expensive

Scheduling rounds

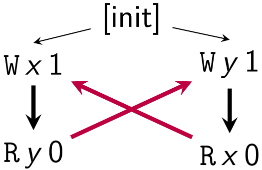
- State-space increases more rapidly
- Optimal exploration
- Fast bound checking

Bounding for weak memory models?

Neither preemptions nor scheduling rounds work.

- Weak behaviors cannot be explained by interleavings.
- Execution graphs can have cycles.

$x := 1$	$y := 1$
$a := y$ reads 0	$b := x$ reads 0



We need a different measure.

- For optimality, TruSt steps must never decrease the measure.
- Revisit steps remove events from the graph.

Bounding for weak memory models

Key observation:

- Removed events were added maximally...
  in an SC fashion with a fixed schedule with no preemptions

Any measure of non-SC-ness works:

- Number of SC-cycles
- Total number of events in SC-cycles
- Number of threads participating in SC-cycles
- Maximal number of events per thread in SC-cycles

Conclusion

Bounding is a nice tool:

- It can make model checking much faster and scale much better.
- It can find all the relevant bugs.

But we need to understand bounding better.

- Especially, for weak memory.