Dynamic Partial Order Reductions for Spinloops

Michalis Kokologiannakis [1] Xiaowei Ren [2] Viktor Vafeiadis [1]

October 2021

[1]Max Planck Institute for Software Systems (MPI-SWS)
[2]The University of British Columbia

Dynamic Partial Order Reduction

Dynamic Partial Order Reduction (DPOR) is a highly effective way to verify *safety properties* of concurrent programs

Dynamic Partial Order Reduction

Dynamic Partial Order Reduction (DPOR) is a highly effective way to verify *safety properties* of concurrent programs

- does not store the visited states

Dynamic Partial Order Reduction

Dynamic Partial Order Reduction (DPOR) is a highly effective way to verify *safety properties* of concurrent programs

- does not store the visited states

DPOR does not work for programs with loops!

Dynamic Partial Order Reduction

Dynamic Partial Order Reduction (DPOR) is a highly effective way to verify *safety properties* of concurrent programs

- does not store the visited states

DPOR does not work for programs with loops!

$\rightsquigarrow$ loops have to be bounded

Bounding is expensive

We want larger bounds to be confident that the program is correct

$\rightsquigarrow$ a bound $b \geq 2$ leads to an exponential blowup

Bounding is expensive

We want larger bounds to be confident that the program is correct

$\rightsquigarrow$ a bound $b \geq 2$ leads to an exponential blowup

Consider a **reader-writer lock** implementation

$\rightsquigarrow$ let's try to verify it using a state-of-the-art DPOR implementation

	$b = 1$	$b = 2$	$b = 3$	$b = 4$
linuxrwlocks	0.02s	26.98s	1366.67s	⏱

Verification time using GenMC (30m timeout)

Bounding is expensive

We want larger bounds to be confident that the program is correct

$\rightsquigarrow$ a bound $b \geq 2$ leads to an exponential blowup

Consider a **reader-writer lock** implementation

$\rightsquigarrow$ let's try to verify it using a state-of-the-art DPOR implementation

	$b = 1$	$b = 2$	$b = 3$	$b = 4$
linuxrwlocks	0.02s	26.98s	1366.67s	⏱

Verification time using GenMC (30m timeout)

DPOR explores the possibility of each loop failing $0, 1, \ldots, b - 1$ times

Solution

Bound all loops with $b = 1$! But when is it sound to do so?

Solution

Bound all loops with $b = 1$! But when is it sound to do so?

$[x = 0]$

```
do
  a := x
while (a = 0)
```

Solution

Bound all loops with $b = 1$! But when is it sound to do so?

$[x = 0]$

$$\begin{array}{l} \textbf{do} \\ \quad a := x \\ \textbf{while}\ (a = 0) \end{array}$$

$\textbf{assume}(x \neq 0)$

↑

Existing techniques

Solution

Bound all loops with $b = 1$! But when is it sound to do so?

$[x = 0]$

do
  $a := x$
while $(a = 0)$

assume$(x \neq 0)$

↑

Existing techniques

What about more complex loops?

Solution

Bound all loops with $b = 1$! But when is it sound to do so?

$[x = 0]$

$$\begin{array}{l} \textbf{do} \\ \quad a := x \\ \textbf{while}\ (a = 0) \end{array}$$

$\textbf{assume}(x \neq 0)$

↑

Existing techniques

What about more complex loops?

↑

This work

SAVER: Spinloop-Aware Verifier

SAVER: Spinloop-Aware Verifier

Effect-free

$[x = 0]$

```
do
  a := x
while (a = 0)
```

Potentially-effect-free

$[x = 0, z = ?]$

```
do
  a := z
  b := CAS(x, 0, 1)
while (a = b)
```

Zero-net-effect

$[x = 0]$

```
while (true)
    a := fetch_add(x, 1)
    if (a = 0) break
    fetch_add(x, −1)
// critical section
fetch_add(x, −1)
```

SAVER: Spinloop-Aware Verifier

Effect-free

$[x = 0]$

```
do
  a := x
while (a = 0)
```

Potentially-effect-free

$[x = 0, z = ?]$

```
do
  a := z
  b := CAS(x, 0, 1)
while (a = b)
```

Zero-net-effect

$[x = 0]$

```
while (true)
    a := fetch_add(x, 1)
    if (a = 0) break
    fetch_add(x, −1)
// critical section
fetch_add(x, −1)
```

Effect-free spinloops

Effect-free loops: iterations lead us to the same state

Effect-free spinloops

Effect-free loops: iterations lead us to the same state

What constitutes an effect-free iteration?

Effect-free spinloops

Effect-free loops: iterations lead us to the same state

What constitutes an effect-free iteration?

- global memory is unmodified

Effect-free spinloops

Effect-free loops: iterations lead us to the same state

What constitutes an effect-free iteration?

- global memory is unmodified
- the loop only assigns at variables dead at the header

Effect-free spinloops

Effect-free loops: iterations lead us to the same state

What constitutes an effect-free iteration?

- global memory is unmodified
- the loop only assigns at variables dead at the header

$[x = 0]$

```
do
  a := x
while (a = 0)
```

Effect-free spinloops

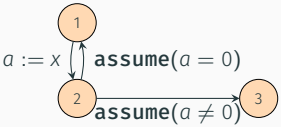
Effect-free loops: iterations lead us to the same state

What constitutes an effect-free iteration?

- global memory is unmodified
- the loop only assigns at variables dead at the header

$[x = 0]$

```
do
  a := x
while (a = 0)
```



Effect-free spinloops

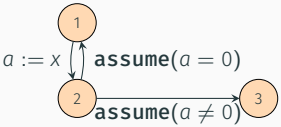
Effect-free loops: iterations lead us to the same state

What constitutes an effect-free iteration?

- global memory is unmodified
- the loop only assigns at variables dead at the header

$[x = 0]$

```
do
  a := x
while (a = 0)
```



We can employ the spin-assume transformation

Effect-free spinloops

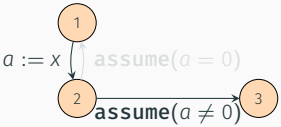
Effect-free loops: iterations lead us to the same state

What constitutes an effect-free iteration?

- global memory is unmodified
- the loop only assigns at variables dead at the header

$[x = 0]$

```
do
  a := x
while (a = 0)
```



We can employ the spin-assume transformation

Effect-free spinloops

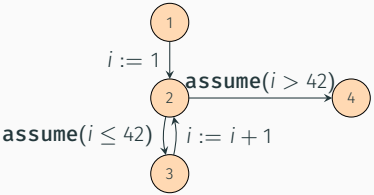
Effect-free loops: iterations lead us to the same state

What constitutes an effect-free iteration?

- global memory is unmodified
- the loop only assigns at variables dead at the header

$[x = 0]$

$\texttt{for}\ (i := 1;\ i \leq 42;\ {+}{+}i)$
$\quad a := x$



Effect-free spinloops

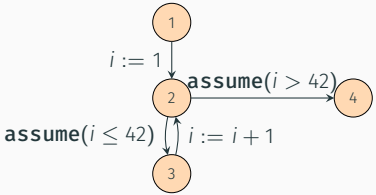
Effect-free loops: iterations lead us to the same state

What constitutes an effect-free iteration?

- global memory is unmodified
- the loop only assigns at variables dead at the header

$[x = 0]$

$\texttt{for}\ (i := 1;\ i \leq 42;\ {+}{+}i)$
$\quad a := x$



Cannot employ spin-assume: modifies variable live at header

Effect-free spinloops

Can we relax the conditions below?

- global memory is unmodified
- the loop only assigns at variables dead at the header

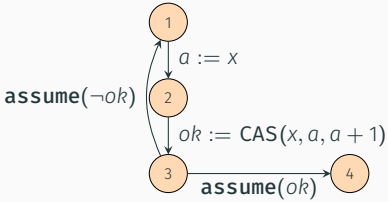
Effect-free spinloops

Can we relax the conditions below?

- global memory is unmodified
- the loop only assigns at variables dead at the header

$[x = 0]$

```
do
    a := x
    ok := CAS(x, a, a + 1)
while (¬ok)
```



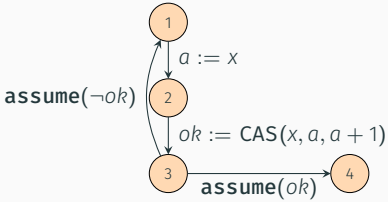
Effect-free spinloops

Can we relax the conditions below?

- global memory is unmodified along looping paths
- the loop only assigns at variables dead at the header

$[x = 0]$

```
do
    a := x
    ok := CAS(x, a, a + 1)
while (¬ok)
```



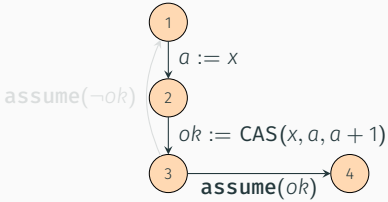
Effect-free spinloops

Can we relax the conditions below?

- global memory is unmodified along looping paths
- the loop only assigns at variables dead at the header

$[x = 0]$

```
do
    a := x
    ok := CAS(x, a, a + 1)
while (¬ok)
```



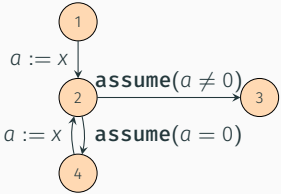
Effect-free spinloops

Can we relax the conditions below?

- global memory is unmodified along looping paths
- the loop only assigns at variables dead at the header

$[x = 0]$

$a := x$
while $(a = 0)$
$a := x$



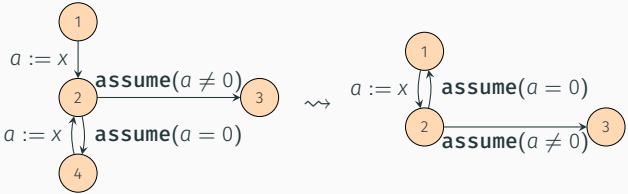
Effect-free spinloops

Can we relax the conditions below?

- global memory is unmodified along looping paths
- the loop only assigns at variables dead at the header modulo bisimilarity

$[x = 0]$

$a := x$
while $(a = 0)$
$a := x$



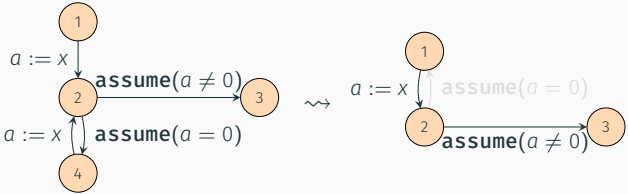
Effect-free spinloops

Can we relax the conditions below?

- global memory is unmodified along looping paths
- the loop only assigns at variables dead at the header modulo bisimilarity

$[x = 0]$

$a := x$
while $(a = 0)$
$a := x$



SAVER: Spinloop-Aware Verifier

Effect-free

$[x = 0]$

```
do
  a := x
while (a = 0)
```

Potentially-effect-free

$[x = 0, z = ?]$

```
do
  a := z
  b := CAS(x, 0, 1)
while (a = b)
```

Zero-net-effect

$[x = 0]$

```
while (true)
    a := fetch_add(x, 1)
    if (a = 0) break
    fetch_add(x, −1)
// critical section
fetch_add(x, −1)
```

SAVER: Spinloop-Aware Verifier

Effect-free

$[x = 0]$

```
do
  a := x
while (a = 0)
```

- Spin-assume
- Bisimilarity/loop rotation

Potentially-effect-free

$[x = 0, z = ?]$

```
do
  a := z
  b := CAS(x, 0, 1)
while (a = b)
```

Zero-net-effect

$[x = 0]$

```
while (true)
    a := fetch_add(x, 1)
    if (a = 0) break
    fetch_add(x, −1)
// critical section
fetch_add(x, −1)
```

SAVER: Spinloop-Aware Verifier

Effect-free

$[x = 0]$

```
do
  a := x
while (a = 0)
```

- Spin-assume
- Bisimilarity/loop rotation

Potentially-effect-free

$[x = 0, z = ?]$

```
do
  a := z
  b := CAS(x, 0, 1)
while (a = b)
```

Zero-net-effect

$[x = 0]$

```
while (true)
    a := fetch_add(x, 1)
    if (a = 0) break
    fetch_add(x, −1)
// critical section
fetch_add(x, −1)
```

Potentially-effect-free spinloops

Potentially-effect-free loops: iterations might lead us to the same state

Potentially-effect-free spinloops

Potentially-effect-free loops: iterations might lead us to the same state

Sometimes this is impossible to know statically

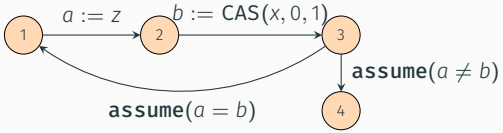
Potentially-effect-free spinloops

Potentially-effect-free loops: iterations might lead us to the same state

Sometimes this is impossible to know statically

$[x = 0, z = ?]$

```
do
    a := z
    b := CAS(x, 0, 1)
while (a = b)
```



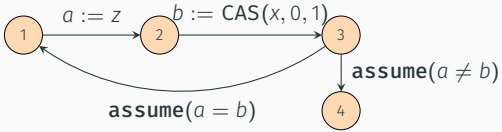
Potentially-effect-free spinloops

Potentially-effect-free loops: iterations might lead us to the same state

Sometimes this is impossible to know statically

$[x = 0, z = ?]$

```
do
  a := z
  b := CAS(x, 0, 1)
while (a = b)
```



We can employ the dynamic spin-assume transformation!

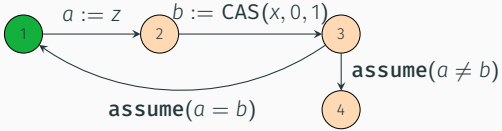
Potentially-effect-free spinloops

Potentially-effect-free loops: iterations might lead us to the same state

Sometimes this is impossible to know statically

$[x = 0, z = ?]$

```
do
  a := z
  b := CAS(x, 0, 1)
while (a = b)
```



We can employ the dynamic spin-assume transformation!

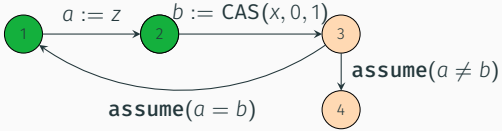
Potentially-effect-free spinloops

Potentially-effect-free loops: iterations might lead us to the same state

Sometimes this is impossible to know statically

$[x = 0, z = ?]$

```
do
    a := z
    b := CAS(x, 0, 1)
while (a = b)
```



We can employ the dynamic spin-assume transformation!

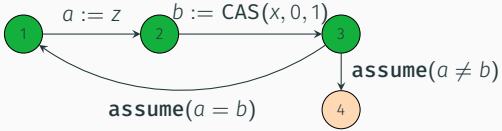
Potentially-effect-free spinloops

Potentially-effect-free loops: iterations might lead us to the same state

Sometimes this is impossible to know statically

$[x = 0, z = ?]$

```
do
  a := z
  b := CAS(x, 0, 1)
while (a = b)
```



We can employ the dynamic spin-assume transformation!

Potentially-effect-free spinloops

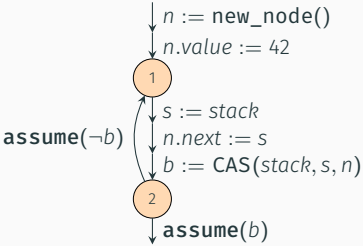
Does this mean we can have stores within a loop?

Potentially-effect-free spinloops

Does this mean we can have stores within a loop?

$[stack = \ldots]$

```
n := new_node() // heap mem
n.value := 42
do
    s := stack
    n.next := s
    b := CAS(stack, s, n)
while (¬b)
```

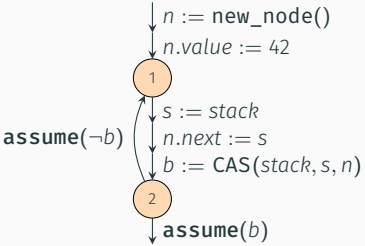


Potentially-effect-free spinloops

Does this mean we can have stores within a loop?

$[stack = \ldots]$

```
n := new_node() // heap mem
n.value := 42
do
    s := stack
    n.next := s
    b := CAS(stack, s, n)
while (¬b)
```



Yes, as long as the stores are not visible to other threads!

SAVER: Spinloop-Aware Verifier

Effect-free

$[x = 0]$

```
do
  a := x
while (a = 0)
```

- Spin-assume
- Bisimilarity/loop rotation

Potentially-effect-free

$[x = 0, z = ?]$

```
do
  a := z
  b := CAS(x, 0, 1)
while (a = b)
```

Zero-net-effect

$[x = 0]$

```
while (true)
    a := fetch_add(x, 1)
    if (a = 0) break
    fetch_add(x, −1)
// critical section
fetch_add(x, −1)
```

SAVER: Spinloop-Aware Verifier

Effect-free

$[x = 0]$

```
do
  a := x
while (a = 0)
```

- Spin-assume
- Bisimilarity/loop rotation

Potentially-effect-free

$[x = 0, z = ?]$

```
do
  a := z
  b := CAS(x, 0, 1)
while (a = b)
```

- Dynamic spin-assume

Zero-net-effect

$[x = 0]$

```
while (true)
    a := fetch_add(x, 1)
    if (a = 0) break
    fetch_add(x, −1)
// critical section
fetch_add(x, −1)
```

SAVER: Spinloop-Aware Verifier

Effect-free

$[x = 0]$

```
do
  a := x
while (a = 0)
```

- Spin-assume
- Bisimilarity/loop rotation

Potentially-effect-free

$[x = 0, z = ?]$

```
do
  a := z
  b := CAS(x, 0, 1)
while (a = b)
```

- Dynamic spin-assume

Zero-net-effect

$[x = 0]$

```
while (true)
    a := fetch_add(x, 1)
    if (a = 0) break
    fetch_add(x, −1)
// critical section
fetch_add(x, −1)
```

Zero-net-effect spinloops

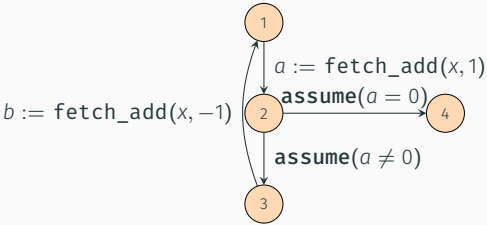
Zero-net-effect loops: iterations with instructions that cancel each other out

Zero-net-effect spinloops

Zero-net-effect loops: iterations with instructions that cancel each other out

$[x = 0]$

```
while (true)
    a := fetch_add(x, 1)
    if (a = 0) break
    fetch_add(x, −1)
// critical section
fetch_add(x, −1)
```

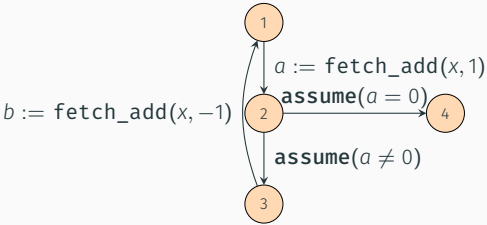


Zero-net-effect spinloops

Zero-net-effect loops: iterations with instructions that cancel each other out

$[x = 0]$

```
while (true)
    a := fetch_add(x, 1)
    if (a = 0) break
    fetch_add(x, −1)
// critical section
fetch_add(x, −1)
```



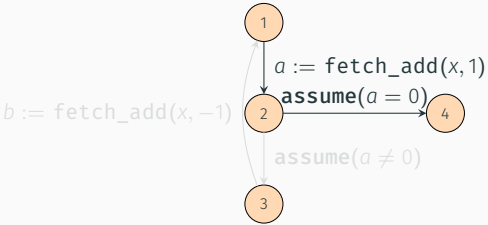
Employ the ZNE-assume transformation

Zero-net-effect spinloops

Zero-net-effect loops: iterations with instructions that cancel each other out

$[x = 0]$

```
while (true)
    a := fetch_add(x, 1)
    if (a = 0) break
    fetch_add(x, −1)
// critical section
fetch_add(x, −1)
```



Employ the ZNE-assume transformation

Zero-net-effect spinloops

ZNE-assume is not always sound!

Zero-net-effect spinloops

ZNE-assume is not always sound!

$[x = 0]$

while $(true)$ $\quad a := \texttt{fetch\_add}(x, 1)$ $\quad$ **if** $(a = 42)$ **break** $\quad \texttt{fetch\_add}(x, -1)$	$b := x$ $c := x$

Zero-net-effect spinloops

ZNE-assume is not always sound!

$[x = 0]$

while $(true)$ $\quad a := \texttt{fetch\_add}(x, 1)$ $\quad$ **if** $(a = 42)$ **break** $\quad \texttt{fetch\_add}(x, -1)$	$b := x$ // 1 $c := x$ // 0

Zero-net-effect spinloops

ZNE-assume is not always sound!

$[x = 0]$

while (true)	
$a :=$ `fetch_add`$(x, 1)$	$b := x$ // 1
if $(a = 42)$ **break**	$c := x$ // 1
`fetch_add`$(x, -1)$	

Zero-net-effect spinloops

ZNE-assume is not always sound!

$[x = 0]$

while $(true)$ $\quad a := \texttt{fetch\_add}(x, 1)$ $\quad$ **if** $(a = 42)$ **break** $\quad \texttt{fetch\_add}(x, -1)$	$b := x$ $c := x$

Need to check ZNE-assume validity dynamically!

SAVER: Spinloop-Aware Verifier

Effect-free

$[x = 0]$

```
do
  a := x
while (a = 0)
```

- Spin-assume
- Bisimilarity/loop rotation

Potentially-effect-free

$[x = 0, z = ?]$

```
do
  a := z
  b := CAS(x, 0, 1)
while (a = b)
```

- Dynamic spin-assume

Zero-net-effect

$[x = y = 0]$

```
while (true)
    a := fetch_add(x, 1)
    if (a = 0) break
    fetch_add(x, −1)
// critical section
fetch_add(x, −1)
```

SAVER: Spinloop-Aware Verifier

Effect-free

$[x = 0]$

```
do
  a := x
while (a = 0)
```

- Spin-assume
- Bisimilarity/loop rotation

Potentially-effect-free

$[x = 0, z = ?]$

```
do
  a := z
  b := CAS(x, 0, 1)
while (a = b)
```

- Dynamic spin-assume

Zero-net-effect

$[x = y = 0]$

```
while (true)
    a := fetch_add(x, 1)
    if (a = 0) break
    fetch_add(x, −1)
// critical section
fetch_add(x, −1)
```

- ZNE-assume
- Dynamic validity checks

SAVER: Spinloop-Aware Verifier

Effect-free

$[x = 0]$

```
do
  a := x
while (a = 0)
```

- Spin-assume
- Bisimilarity/loop rotation

Potentially-effect-free

$[x = 0, z = ?]$

```
do
  a := z
  b := CAS(x, 0, 1)
while (a = b)
```

- Dynamic spin-assume

Zero-net-effect

$[x = y = 0]$

```
while (true)
    a := fetch_add(x, 1)
    if (a = 0) break
    fetch_add(x, −1)
// critical section
fetch_add(x, −1)
```

- ZNE-assume
- Dynamic validity checks

We can apply a different transformation to each backedge!

Putting everything together: Michael-Scott queue

$[head = \ldots, tail = \ldots]$

```
success := false
while (¬success)
    h := head
    t := tail
    n := next[h]
    h' := head
    if (h ≠ h') continue
    if (h = t)
        if (n) break
        CAS(tail, t, n)
    else
        success := CAS(head, h, n)
```

Putting everything together: Michael-Scott queue

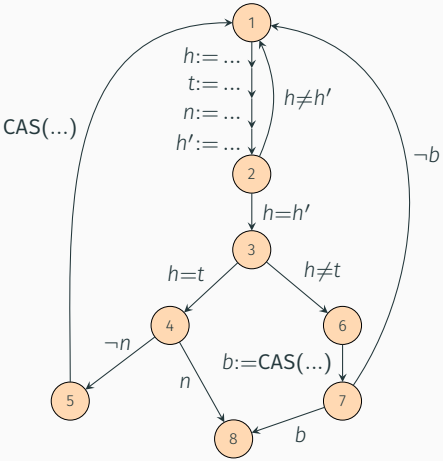
$[head = \ldots, tail = \ldots]$

```
while (true)  // loop rotation
    h := head
    t := tail
    n := next[h]
    h' := head
    if (h ≠ h') continue
    if (h = t)
        if (n) break
        CAS(tail, t, n)
    else
        b := CAS(head, h, n)
        if (b) break
```

Putting everything together: Michael-Scott queue

$[head = \ldots, tail = \ldots]$

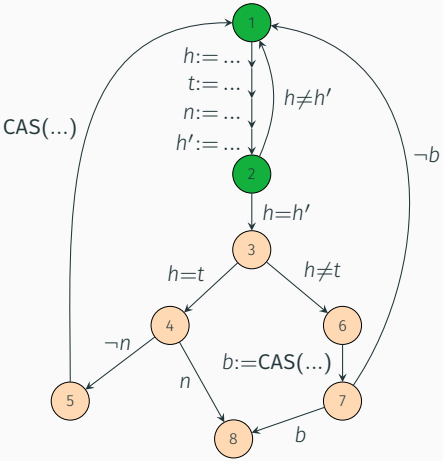
```
while (true)  //loop rotation
    h := head
    t := tail
    n := next[h]
    h' := head
    if (h ≠ h') continue
    if (h = t)
        if (n) break
        CAS(tail, t, n)
    else
        b := CAS(head, h, n)
        if (b) break
```



Putting everything together: Michael-Scott queue

$[head = \ldots, tail = \ldots]$

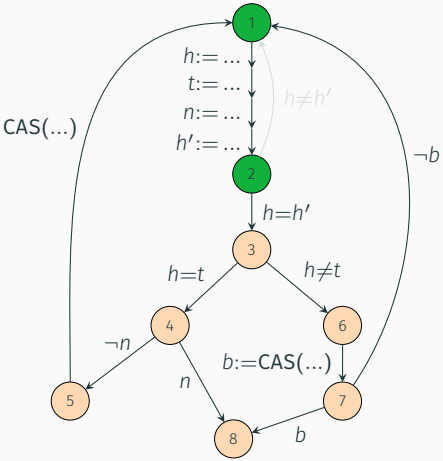
```
while (true)  // loop rotation
    h := head
    t := tail
    n := next[h]
    h' := head
    if (h ≠ h') continue
    if (h = t)
        if (n) break
        CAS(tail, t, n)
    else
        b := CAS(head, h, n)
        if (b) break
```



Putting everything together: Michael-Scott queue

$[head = \ldots, tail = \ldots]$

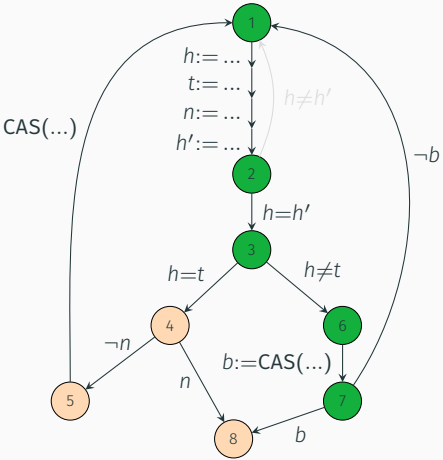
```
while (true)  // loop rotation
    h := head
    t := tail
    n := next[h]
    h' := head
    if (h ≠ h') continue // SA
    if (h = t)
        if (n) break
        CAS(tail, t, n)
    else
        b := CAS(head, h, n)
        if (b) break
```



Putting everything together: Michael-Scott queue

$[head = \ldots, tail = \ldots]$

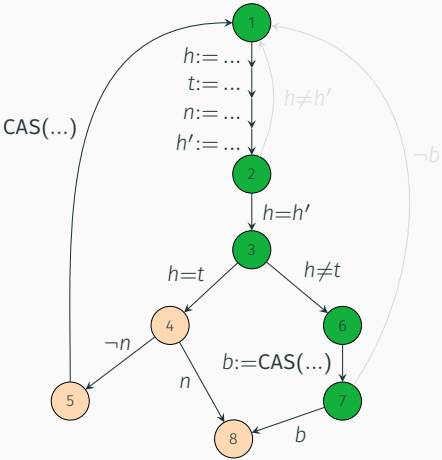
```
while (true)  // loop rotation
    h := head
    t := tail
    n := next[h]
    h' := head
    if (h ≠ h')  continue // SA
    if (h = t)
        if (n)  break
        CAS(tail, t, n)
    else
        b := CAS(head, h, n)
        if (b)  break
```



Putting everything together: Michael-Scott queue

$[head = \ldots, tail = \ldots]$

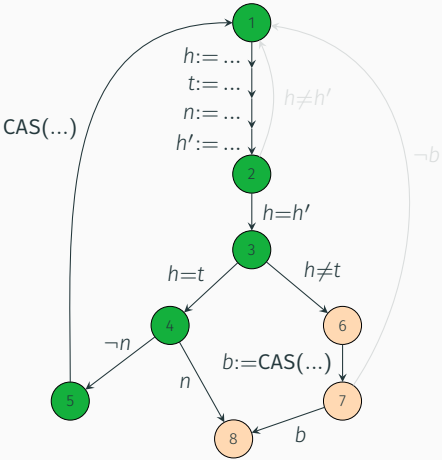
```
while (true)  // loop rotation
    h := head
    t := tail
    n := next[h]
    h' := head
    if (h ≠ h')  continue // SA
    if (h = t)
        if (n)  break
        CAS(tail, t, n)
    else
        b := CAS(head, h, n)
        if (b)  break // SA
```



Putting everything together: Michael-Scott queue

$[head = \ldots, tail = \ldots]$

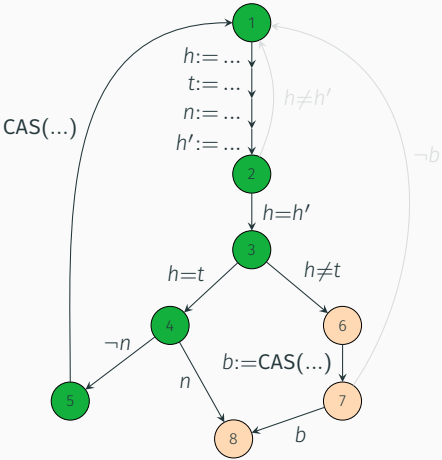
```
while (true)  // loop rotation
    h := head
    t := tail
    n := next[h]
    h' := head
    if (h ≠ h')  continue // SA
    if (h = t)
        if (n)  break
        CAS(tail, t, n)
    else
        b := CAS(head, h, n)
        if (b)  break // SA
```



Putting everything together: Michael-Scott queue

$[head = \ldots, tail = \ldots]$

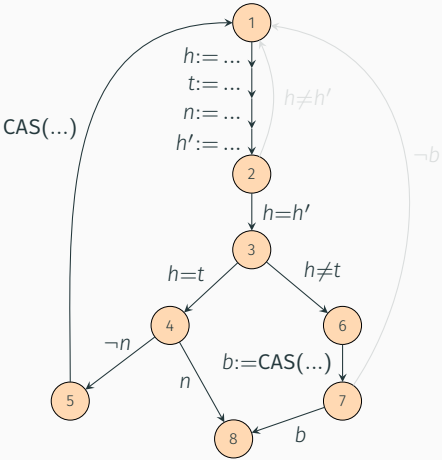
```
while (true)  // loop rotation
    h := head
    t := tail
    n := next[h]
    h' := head
    if (h ≠ h')  continue // SA
    if (h = t)
        if (n)  break
        CAS(tail, t, n) // DSA
    else
        b := CAS(head, h, n)
        if (b)  break // SA
```



Putting everything together: Michael-Scott queue

$[head = \ldots, tail = \ldots]$

```
while (true)  // loop rotation
    h := head
    t := tail
    n := next[h]
    h' := head
    if (h ≠ h') continue // SA
    if (h = t)
        if (n) break
        CAS(tail, t, n) // DSA
    else
        b := CAS(head, h, n)
        if (b) break // SA
```



Results

Realistic benchmarks

	GenMC$_{\setminus S}$	GenMC	Saver		
	Execs	*Execs*	*Execs*	*Time*	*Trans*
linuxrwlocks(3)	14059037	38033	24	.04	B, S, Z
linuxrwlocks(4)	⏰	⏰	1060	.36	B, S, Z
chase-lev(5)	17367	17367	3835	.20	S
chase-lev(6)	778581	778581	41055	2.39	S
treiber-stack(3)	426	426	18	.10	S, D
treiber-stack(4)	1546168	1546168	484	.61	S, D
ttaslock(3)	11031	11031	162	.10	S, D
ttaslock(4)	⏰	⏰	20760	2.46	S, D
twalock(3)	1338	1338	96	.10	S
twalock(4)	1018872	1018872	6144	.72	S
ms-queue(3)	1389	1389	75	.09	L, S, D
ms-queue(4)	⏰	⏰	10662	28.13	L, S, D

Realistic benchmarks

	GenMC$_{\setminus S}$	GenMC	Saver		
	Execs	*Execs*	*Execs*	*Time*	*Trans*
linuxrwlocks(3)	14059037	38033	24	.04	B, S, Z
linuxrwlocks(4)	⏰	⏰	1060	.36	B, S, Z
chase-lev(5)	17367	17367	3835	.20	S
chase-lev(6)	778581	778581	41055	2.39	S
treiber-stack(3)	426	426	18	.10	S, D
treiber-stack(4)	1546168	1546168	484	.61	S, D
ttaslock(3)	11031	11031	162	.10	S, D
ttaslock(4)	⏰	⏰	20760	2.46	S, D
twalock(3)	1338	1338	96	.10	S
twalock(4)	1018872	1018872	6144	.72	S
ms-queue(3)	1389	1389	75	.09	L, S, D
ms-queue(4)	⏰	⏰	10662	28.13	L, S, D

Conclusion

SAVER: a Spinloop-Aware Verifier

- novel extensions to DPOR
  - spin-assume
  - bisimilarity/loop rotation
  - dynamic-spin-assume
  - ZNE-assume
- works for a variety of memory models
- exponentially fewer executions than state-of-the-art

`https://github.com/MPI-SWS/genmc`