VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models

Ming Fu[⋆†]
ming.fu@huawei.com

Joint work with Jonas Oberhauser[⋆†], Rafael Chehab[⋆†], Diogo Behrens[⋆†], Antonio Paolillo[⋆†], Lilith Oberhauser[⋆†], Koustubha Bhat[⋆†], Yuzhong Wen[†], Haibo Chen[†], Jaeho Kim[⋆†], Viktor Vafeiadis[*]

[⋆]Huawei Dresden Research Center
[†]Huawei OS Kernel Lab

[*]MPI-SWS

December 16, 2020

Agenda

- **Motivation**
- **Challenges**
- **VSync Framework**
  - **Adaptive Linear Relaxation (ALR)**
  - **Await Model Checking (AMC)**
- **Evaluation**
- **Summary and Outlook**

Agenda

- **Motivation**
- Challenges
- VSync Framework
  - Adaptive Linear Relaxation (ALR)
  - Await Model Checking (AMC)
- Evaluation
- Summary and Outlook

ARM is everywhere...

- 90% mobile-processor market share (2020)[1]
  your smartphone has one...
- 40% of produced chips contain Arm (2017)[2]
- And growing...

[1] https://seekingalpha.com/article/4233721-arm-holdings-unknown-british-tech-firm-will-drive-softbanks-future-growth

[2] https://group.softbank/system/files/pdf/ir/presentations/2019/Arm_SB_Q2_2018_Roadshow_Slides_FINAL_en.pdf

ARM is everywhere. . .

Huawei Unveils Industry's Highest-Performance ARM-based CPU

Bringing Global Computing Power to Next Level

Jan 07, 2019

[Shenzhen, China, January 7, 2019] Today, Huawei announced the industry's highest-performance Advanced RISC Machine (ARM)-based CPU. Called Huawei Kunpeng 920, the new CPU is designed to boost the development of computing in big data, distributed storage, and ARM-native application scenarios. Huawei will join with industry players to advance the ARM industry and foster an open, collaborative, and win-win ecosystem, taking computing performance to new heights.

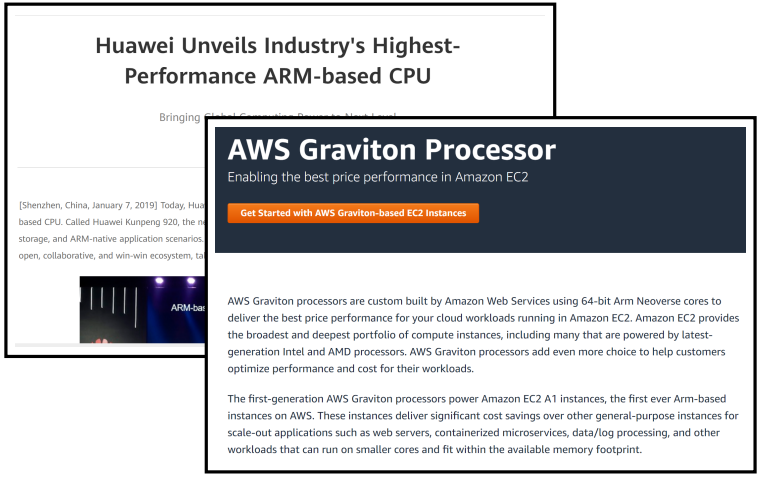
20)[1]

017)[2]

[1] https://seekingalpha.com/article/4233721-arm-holdings-unknown-british-tech-firm-will-drive-softbanks-future-growth

[2] https://group.softbank/system/files/pdf/ir/presentations/2019/Arm_SB_Q2_2018_Roadshow_Slides_FINAL_en.pdf

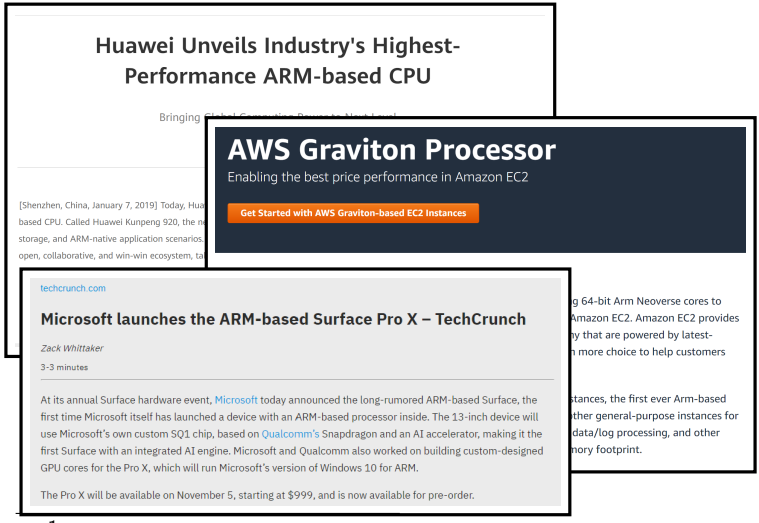
ARM is everywhere. . .



[1] https://seekingalpha.com/article/4233721-arm-holdings-unknown-british-tech-firm-will-drive-softbanks-future-growth

[2] https://group.softbank/system/files/pdf/ir/presentations/2019/Arm_SB_Q2_2018_Roadshow_Slides_FINAL_en.pdf

ARM is everywhere. . .



[1] https://seekingalpha.com/article/4233721-arm-holdings-unknown-british-tech-firm-will-drive-softbanks-future-growth

[2] https://group.softbank/system/files/pdf/ir/presentations/2019/Arm_SB_Q2_2018_Roadshow_Slides_FINAL_en.pdf

ARM is everywhere. . .

[1] https://seekingalpha.com/article/4233721-arm-holdings-unknown-british-tech-firm-will-drive-softbanks-future-growth

[2] https://group.softbank/system/files/pdf/ir/presentations/2019/Arm_SB_Q2_2018_Roadshow_Slides_FINAL_en.pdf

And RISC-V might soon be everywhere too...

- "**By 2021, a billion cores** are expected to ship **using the RISC-V** architecture"

  Zvonimir Bandic, foundation board member and tech director at Western Digital[3]

- "**Companies can build their own core to fit their own needs**. They can have greater visibility into how the core runs, and even develop their own security features."

  Krste Asanović, Berkeley professor (with Patterson, RISC-V creators)[4]

[3] https://venturebeat.com/2019/12/11/risc-v-grows-globally-as-an-alternative-to-arm-and-its-license-fees

[4] Will RISC-V Revolutionize Computing? – Samuel Greengard, Communications of the ACM, May 2020, https://cacm.acm.org/magazines/2020/5/244325-will-risc-v-revolutionize-computing/

But the challenges come along...

- **ARM and RISC-V are different than x86 for multicore concurrency!**
  - Goal: Performance and power efficiency
  - Approach: Weak Memory Model (WMM)
  - Problem: WMM behavior is hard to control

But the challenges come along...

- **ARM and RISC-V are different than x86 for multicore concurrency!**
  - Goal: Performance and power efficiency
  - Approach: Weak Memory Model (WMM)
  - Problem: WMM behavior is hard to control
- **Good news: most programs are still correct**
  - Employ spinlocks and mutexes for synchronization
  - Do not use ad-hoc synchronization with atomics

But the challenges come along...

- **ARM and RISC-V are different than x86 for multicore concurrency!**
  - Goal: Performance and power efficiency
  - Approach: Weak Memory Model (WMM)
  - Problem: WMM behavior is hard to control
- **Good news: most programs are still correct**
  - Employ spinlocks and mutexes for synchronization
  - Do not use ad-hoc synchronization with atomics
- **Bad news: spinlocks/mutexes can be incorrect!**
  - Lie on the critical path
  - Most of them designed with x86 in mind
  - Often too strong or too weak barriers
  - Seldom just right!

Meanwhile in the academic community. . .

Meanwhile in the academic community...

Recently-published synchronization primitives:

Primitive	Authors	Venue'Year	Discuss WMM?
ShflLock	Kashyap et al.	SOSP'19	Authors do not mention memory models
CNA lock	Dice and Kogan	EuroSys'19	"actual implementation uses fences where necessary"
TWA lock	Dice and Kogan	EuroPar'19	Authors only talk about x86 and SPARC TSO
HMCS lock	Chabbi et al.	PPoPP'15	Yes, authors provide barriers, but they are insufficient

WMM stories in open-source projects

Potential hang: DPDK MCS lock

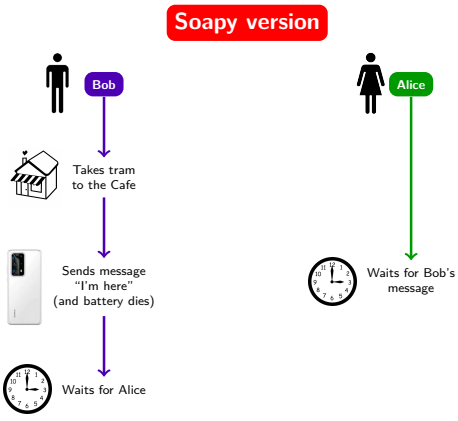
The lock had one missing release barrier. An ARM engineer replied to our patch: "Unfortunately, memory ordering questions are hard topics. I have been discussing this internally [...], hope to conclude soon." More 3 months to accept the 1-line patch.

Mutual exclusion bug: seL4 CLH lock

The seL4 microkernel is a flagship of formal verification. The big kernel lock implementation, however, was not verified and missed one acquire-release barrier.

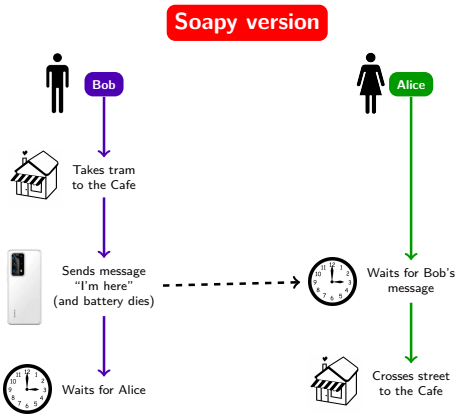
Why is WMM so hard?

A tale of a date



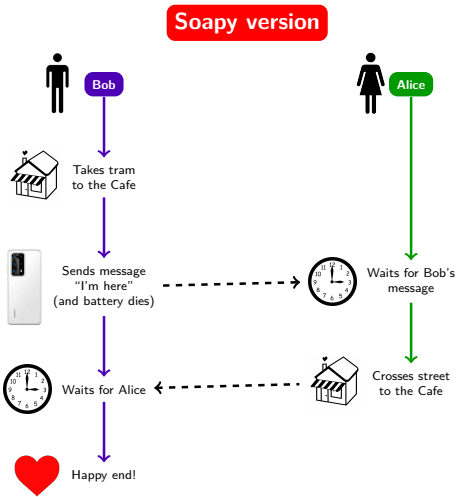
Why is WMM so hard?

A tale of a date



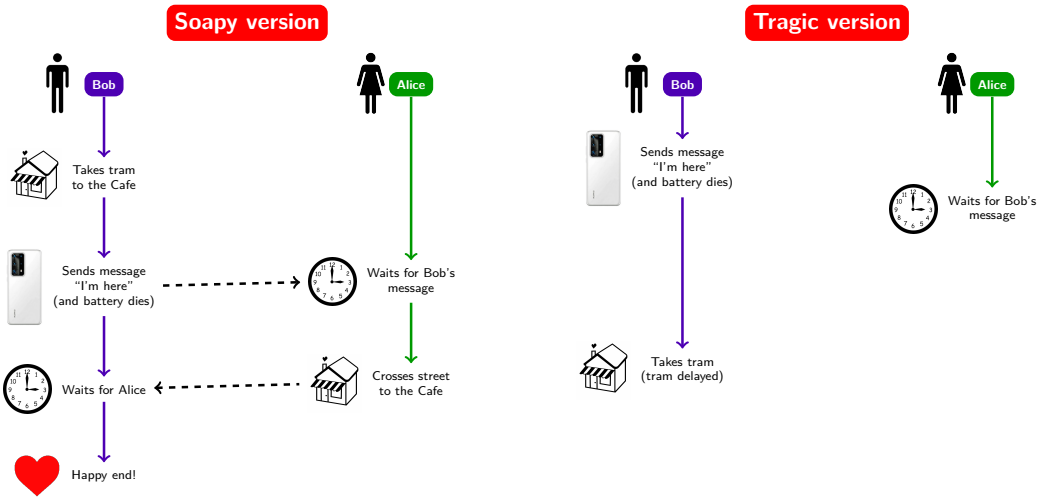
Why is WMM so hard?

A tale of a date



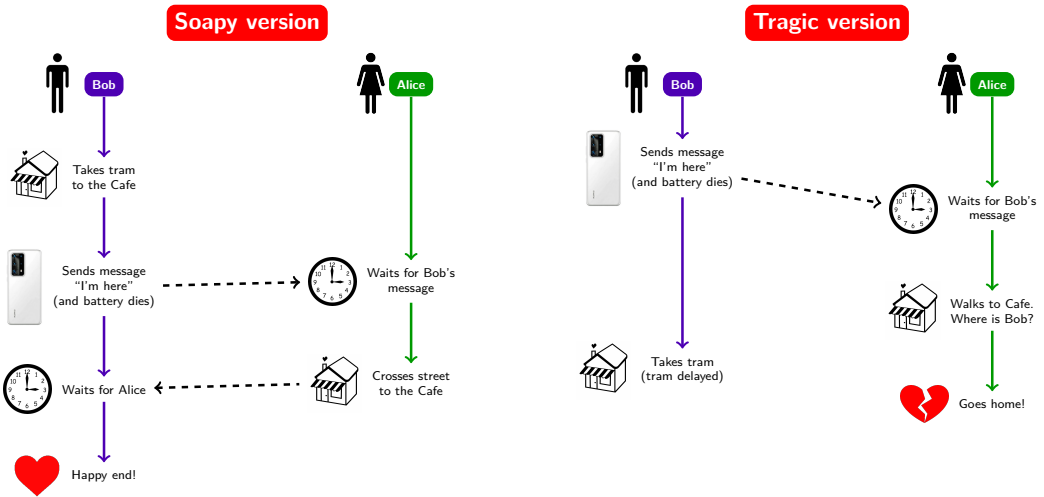
Why is WMM so hard?

A tale of a date



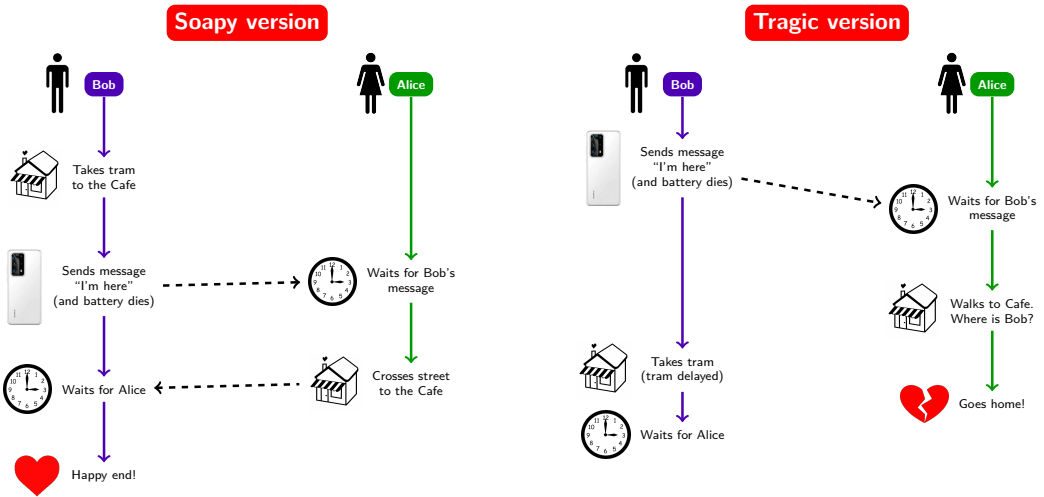
Why is WMM so hard?

A tale of a date



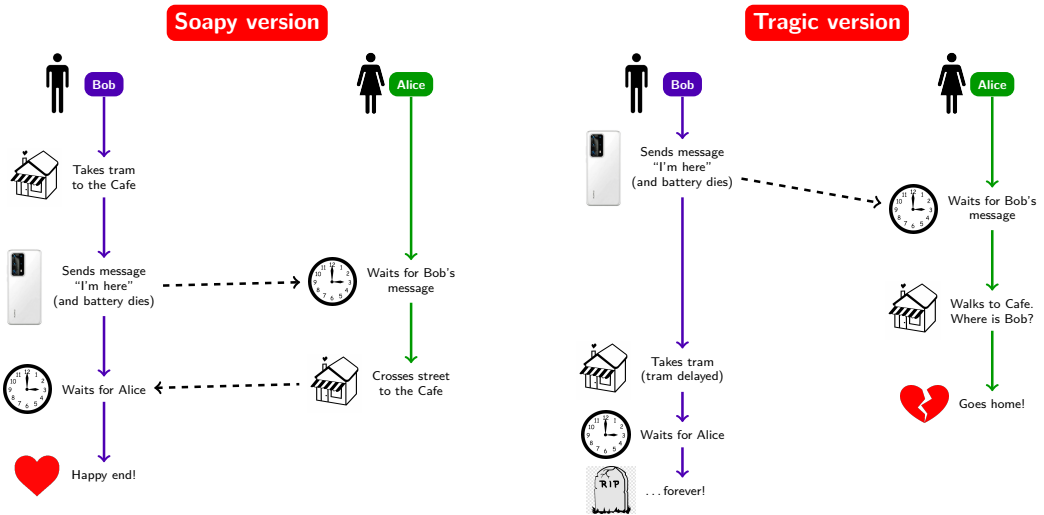
Why is WMM so hard?

A tale of a date

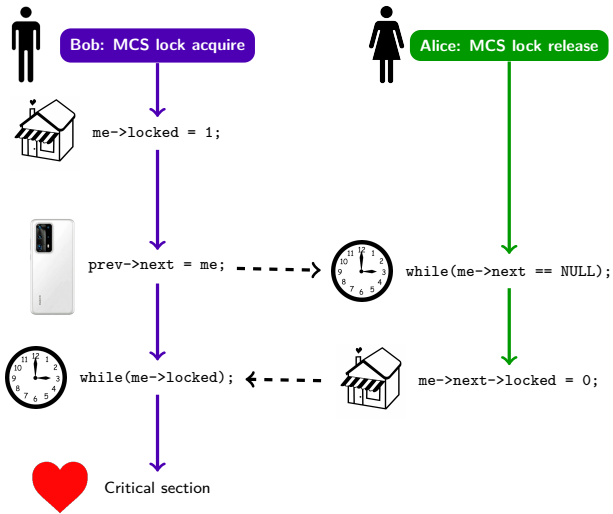


Why is WMM so hard?

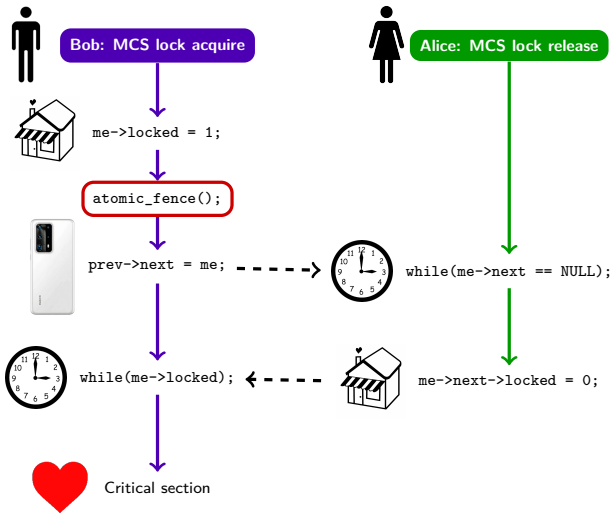
A tale of a date



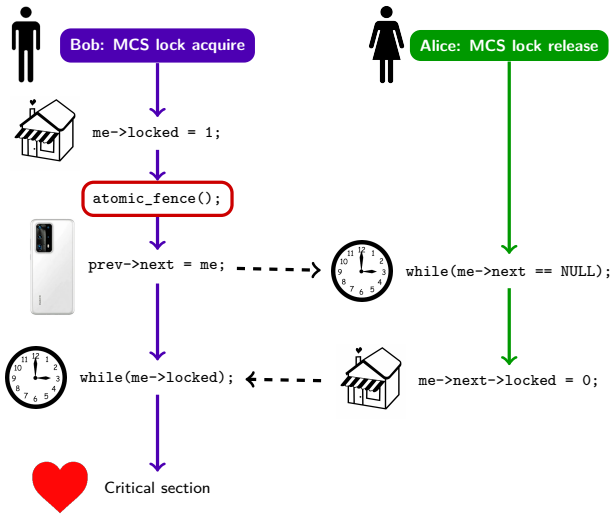
Mapping the tale to DPDK MCS lock bug



Mapping the tale to DPDK MCS lock bug



Mapping the tale to DPDK MCS lock bug



- Compiler and hardware optimize executions by default
- **Memory barriers/fences** control optimizations
- But be careful: Barriers/fences **degrade performance**

Some solutions. . .

Solution 1: overprotection

The musl project conservatively overprotects the code by adding many unnecessary strong barriers. Depending on the hardware, the additional barriers may hurt performance.

Solution 2: expert-optimization

Over the course of 6 years, the Linux qspinlock was optimized by highly-skilled engineers. Still, one bug kept dormant during 12 releases.

CAN WE HAVE AN AUTOMATED SOLUTION?

Agenda

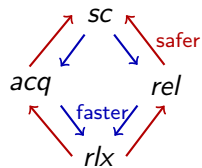
- Motivation
- **Challenges**
- VSync Framework
  - Adaptive Linear Relaxation (ALR)
  - Await Model Checking (AMC)
- Evaluation
- Summary and Outlook

User-provided implementation

```
/* lock acquire */
do {
  while(atomic_read(&lock) != 0);
} while(atomic_xchg(&lock, 1) != 0);

/* critical section */
x++;

/* lock release */
atomic_write(&lock, 0);
```

User-provided implementation

```
/* lock acquire */
do {
  while(atomic_read(&lock) != 0);
} while(atomic_xchg(&lock, 1) != 0);

/* critical section */
x++;

/* lock release */
atomic_write(&lock,
```



Each of these operation has **a barrier attached**!

VSYNC-atomics	RISC-V	ARMv8	x86
atomic_xchg	amoswap.w.aq.rl.sc	LDAXR;STLXR	XCHG
atomic_xchg_rel	amoswap.w.rl	LDXR;STLXR	XCHG
atomic_xchg_acq	amoswap.w.aq	LDAXR;STXR	XCHG
atomic_xchg_rlx	amoswap.w	LDXR;STXR	XCHG
atomic_read	fence [rw,rw]; lw; fence [r,rw]	LDAR	MOV
atomic_read_acq	lw; fence [r,rw]	LDAR	MOV
atomic_read_rlx	lw	LDR	MOV
atomic_write	fence [rw,w]; sw; fence [rw,rw]	STLR	MOV;MFENCE
atomic_write_rel	fence [rw,w]; sw	STLR	MOV
atomic_write_rlx	sw	STR	MOV
atomic_fence	fence [rw,rw]	DMB ISH	MFENCE
atomic_fence_acq	fence [r,rw]	DMB ISHLD	NOP
atomic_fence_rel	fence [rw,w]	DMB ISH	NOP
atomic_fence_rlx	nop	NOP	NOP

User-provided implementation

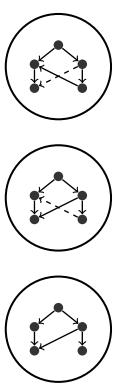
```
/* lock acquire */
do {
  while(atomic_read(&lock) != 0);
} while(atomic_xchg(&lock, 1) != 0);

/* critical section */
x++;

/* lock release */
atomic_write(&lock, 0);
```

Challenge: How to verify safety and liveness?

User-provided implementation

```
/* lock acquire */
do {
  while(atomic_read(&lock) != 0);
} while(atomic_xchg(&lock, 1) != 0);

/* critical section */
x++;

/* lock release */
atomic_write(&lock, 0);
```

Challenge: How to verify safety and liveness?

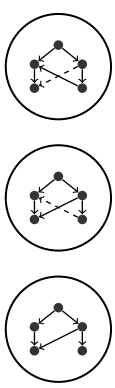


Let's use model checking (MC)!

- Pratical for automation ($\neq$ proving)
- Cover all executions ($\neq$ testing)

User-provided implementation

```
/* lock acquire */
do {
  while(atomic_read(&lock) != 0);
} while(atomic_xchg(&lock, 1) != 0);

/* critical section */
x++;

/* lock release */
atomic_write(&lock, 0);
```



Let's use model checking (MC)!

- Pratical for automation ($\neq$ proving)
- Cover all executions ($\neq$ testing)

Challenge: How to verify safety and liveness?

- Stateful MCs **do not scale with WMM**, eg, Rmem, Power2SC, TLA/TLC
- Stateless MCs **cannot detect non-termination**, eg, GenMC, Nidhug

User-provided implementation

```
/* lock acquire */
do {
  while(atomic_read(&lock) != 0);
} while(atomic_xchg(&lock, 1) != 0);

/* critical section */
x++;

/* lock release */
atomic_write(&lock, 0);
```

Maximally-relaxed implementation

```
/* lock acquire */
do {
  while(atomic_read_rlx(&lock) != 0);
} while(atomic_xchg_acq(&lock, 1) != 0);

/* critical section */
x++;

/* lock release */
atomic_write_rel(&lock, 0);
```

Challenge: How to verify safety and liveness?

Challenge: How to optimize the barriers?

- Stateful MCs **do not scale with WMM**, eg, Rmem, Power2SC, TLA/TLC
- Stateless MCs **cannot detect non-termination**, eg, GenMC, Nidhug

User-provided implementation

```
/* lock acquire */
do {
  while(atomic_read(&lock) != 0);
} while(atomic_xchg(&lock, 1) != 0);

/* critical section */
x++;

/* lock release */
atomic_write(&lock, 0);
```

automatically

Maximally-relaxed implementation

```
/* lock acquire */
do {
  while(atomic_read_rlx(&lock) != 0);
} while(atomic_xchg_acq(&lock, 1) != 0);

/* critical section */
x++;

/* lock release */
atomic_write_rel(&lock, 0);
```

Challenge: How to verify safety and liveness?

- Stateful MCs **do not scale with WMM**, eg, Rmem, Power2SC, TLA/TLC
- Stateless MCs **cannot detect non-termination**, eg, GenMC, Nidhug

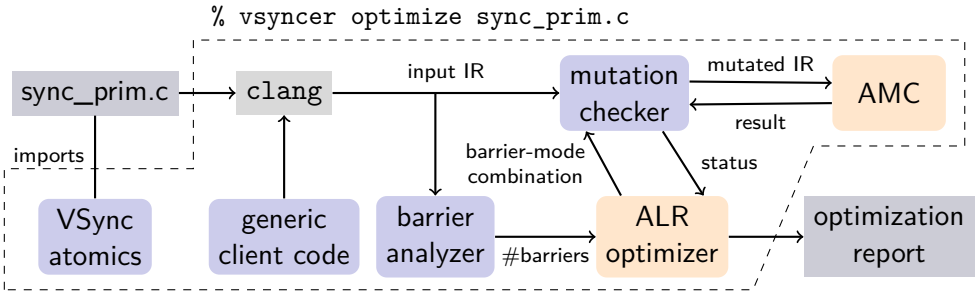
Challenge: How to optimize the barriers?

- **Search space explosion!**
- TTAS lock, 3 positions, 4 modes: $4^3 = 64$ combinations
- Linux qspinlock: $4^{26} = 2^{52}$ combinations
- Brute-force or breadth-first-search won't work

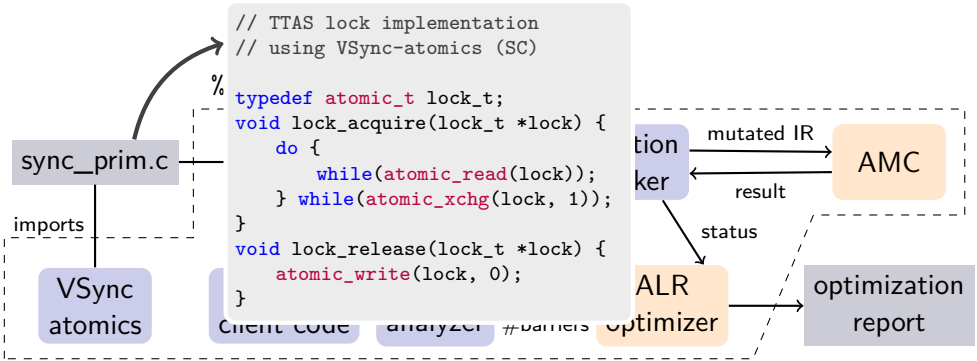
Agenda

- Motivation
- Challenges
- **VSync Framework**
  - **Adaptive Linear Relaxation (ALR)**
  - **Await Model Checking (AMC)**
- Evaluation
- Summary and Outlook

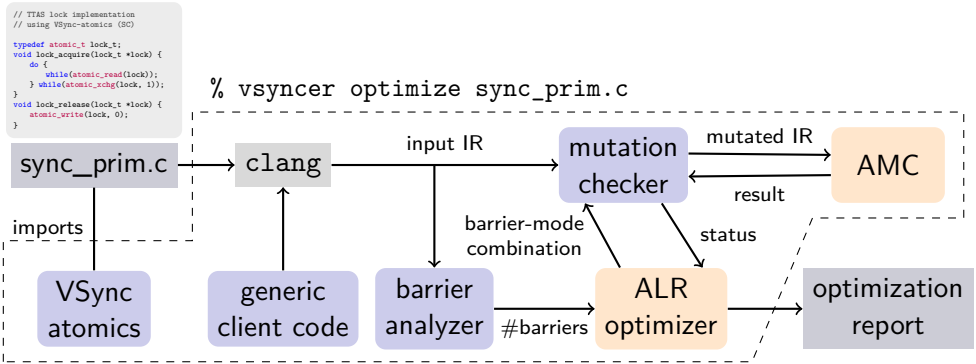
VSync Framework



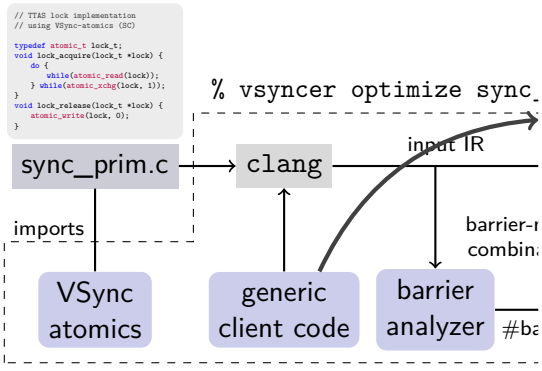
VSync Framework



VSync Framework



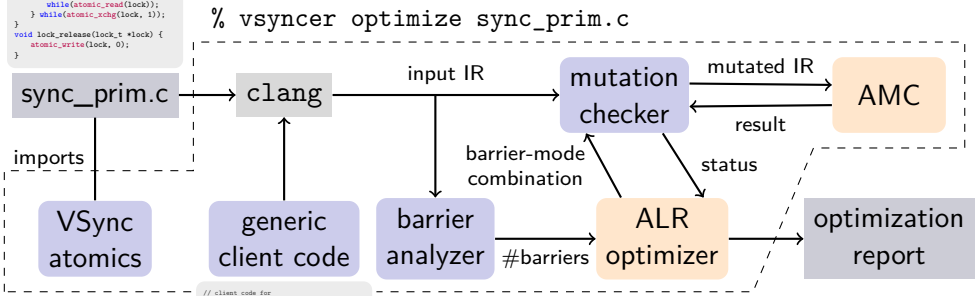
VSync Framework




```c
// client code for
// synchronization primitives

lock_t lock;
int v = 0; // shared state
void run(int id) {
    lock_acquire(&lock);
    v++; // critical sec.
    lock_release(&lock);
}

void main() {
    pthread_t t[N]; // N == 3
    for (int i=0; i < N; i++)
        pthread_create(&t[i], 0, run, i);
    for (int i=0; i < N; i++)
        pthread_join(t[i], 0);
    assert (v == N); // check of ME
}
```

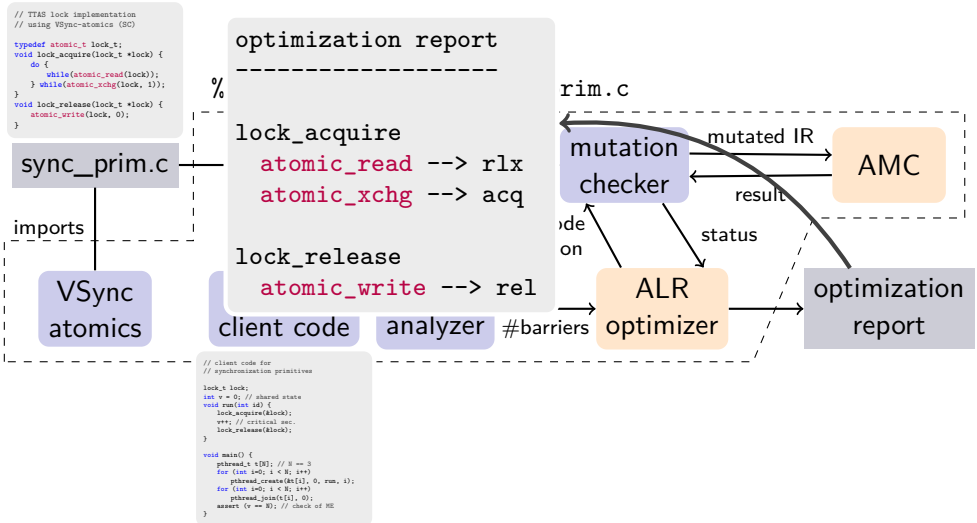
VSync Framework

```
// TTAS lock implementation
// using VSync-atomics (SC)

typedef atomic_t lock_t;
void lock_acquire(lock_t *lock) {
    do {
        while(atomic_read(lock));
    } while(atomic_xchg(lock, 1));
}
void lock_release(lock_t *lock) {
    atomic_write(lock, 0);
}
```

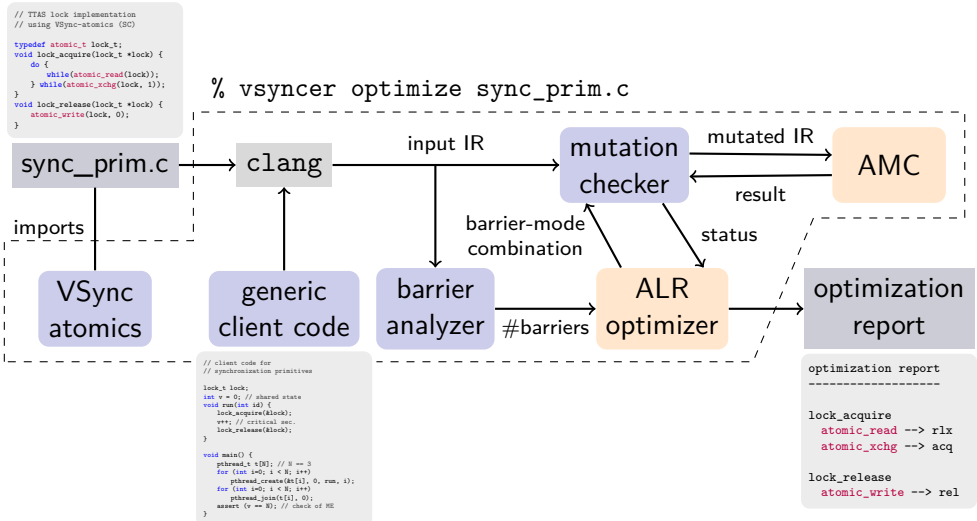



```
// client code for
// synchronization primitives

lock_t lock;
int v = 0; // shared state
void run(int id) {
    lock_acquire(&lock);
    v++; // critical sec.
    lock_release(&lock);
}

void main() {
    pthread_t t[N]; // N == 3
    for (int i=0; i < N; i++)
        pthread_create(&t[i], 0, run, i);
    for (int i=0; i < N; i++)
        pthread_join(t[i], 0);
    assert (v == N); // check of ME
}
```

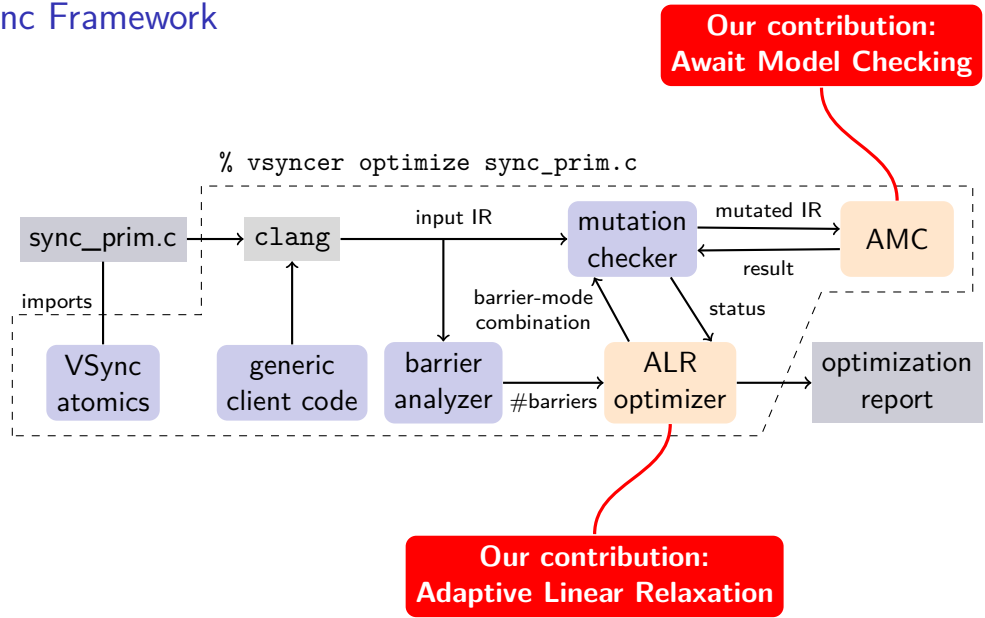
VSync Framework



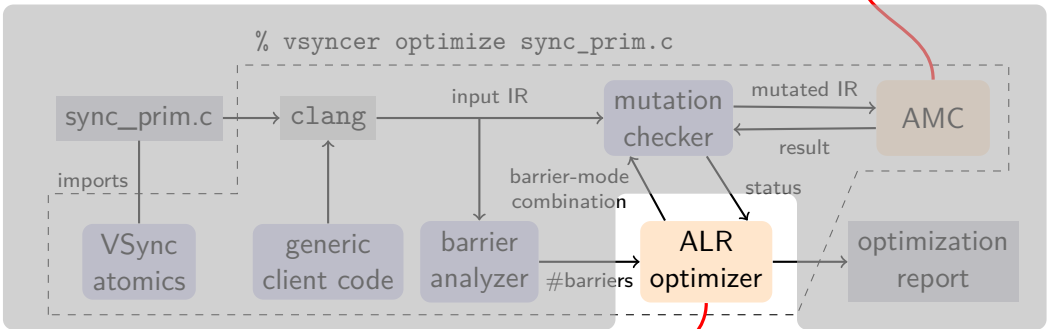
VSync Framework



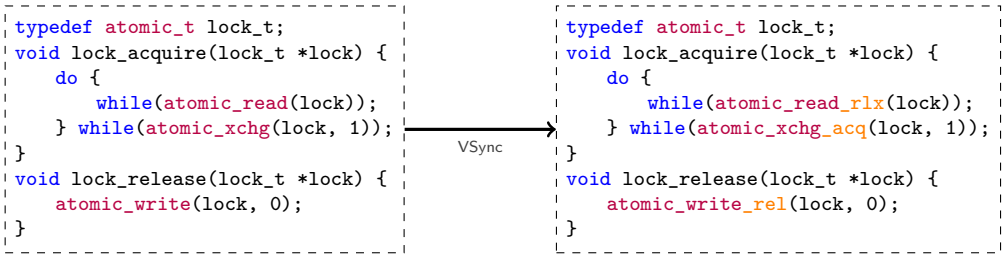
VSync Framework



VSync Framework



Optimization goal



combination $[sc, sc, sc]$ combination $[rlx, acq, rel]$

Linear Relaxation

```
function LR()
    b ← [sc, ..., sc]
    for i ∈ [1, ..., |b|] do
        for m ∈ [rlx, acq, rel, sc] do
            b[i] ← m
            if b[i] = sc ∨ check(b)
              then
                break
    return b
```

Linear Relaxation

```
function LR()
    b ← [sc, ..., sc]
    for i ∈ [1, ..., |b|] do
        for m ∈ [rlx, acq, rel, sc] do
            b[i] ← m
            if b[i] = sc ∨ check(b)
              then
                break
    return b
```

iterations ↓

$b[1]$	$b[2]$	$b[3]$	check(b)
rlx	*sc*	*sc*	yes
rlx	*rlx*	*sc*	no
rlx	*acq*	*sc*	yes
rlx	*acq*	*rlx*	no
rlx	*acq*	*acq*	no
rlx	*acq*	*rel*	yes

Linear Relaxation

```
function LR()
    b ← [sc, ..., sc]
    for i ∈ [1, ..., |b|] do
        for m ∈ [rlx, acq, rel, sc] do
            b[i] ← m
            if b[i] = sc ∨ check(b)
              then
                break
    return b
```

iterations ↓

$b[1]$	$b[2]$	$b[3]$	check(b)
rlx	*sc*	*sc*	yes
rlx	*rlx*	*sc*	no
rlx	*acq*	*sc*	yes
rlx	*acq*	*rlx*	no
rlx	*acq*	*acq*	no
rlx	*acq*	*rel*	yes

Monotonicity insight:
relaxing an already incorrect barrier combination can never produce a correct one.

Adaptive Linear Relaxation (ALR)

How long to optimize Linux qspinlock?

$$T_o \approx C_{\lightning} * t_{\lightning} + C_{\checkmark} * t_{\checkmark}$$

Adaptive Linear Relaxation (ALR)

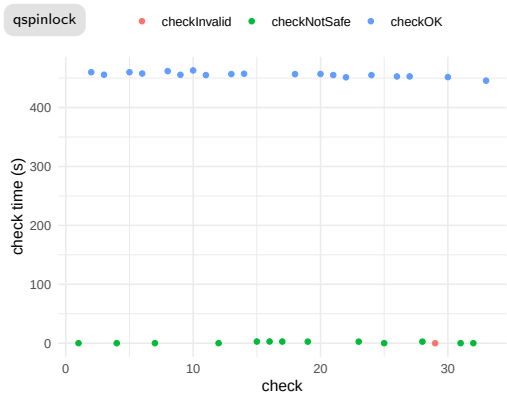
How long to optimize Linux qspinlock?

$$
\begin{aligned}
T_o &\approx C_{\text{✗}} * t_{\text{✗}} + C_{\text{✓}} * t_{\text{✓}} \\
&\approx 14 * t_{\text{✗}} + 19 * t_{\text{✓}}
\end{aligned}
$$

Adaptive Linear Relaxation (ALR)

How long to optimize Linux qspinlock?

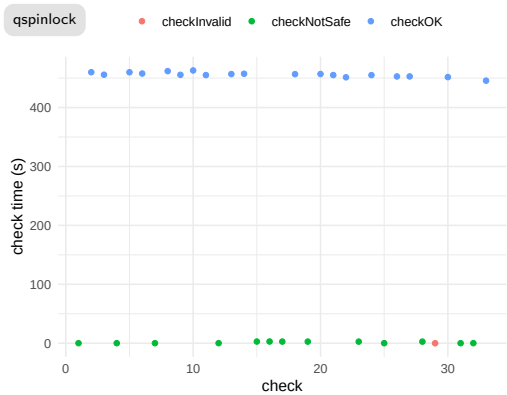
$$T_o \approx C_{\text{✗}} * t_{\text{✗}} + C_{\text{✓}} * t_{\text{✓}}$$
$$\approx 14 * t_{\text{✗}} + 19 * t_{\text{✓}}$$



Adaptive Linear Relaxation (ALR)

How long to optimize Linux qspinlock?

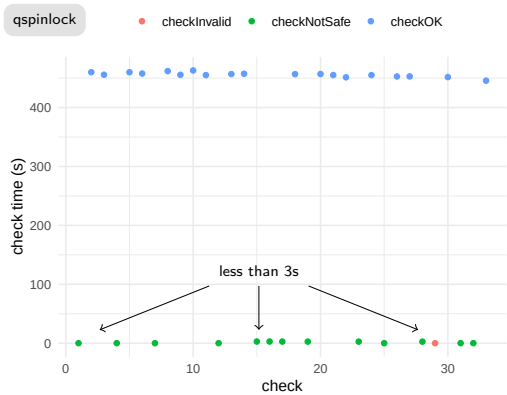
$$
\begin{aligned}
T_o &\approx C_{\lightning} * t_{\lightning} + C_{\checkmark} * t_{\checkmark} \\
&\approx 14 * t_{\lightning} + 19 * t_{\checkmark} \\
&\approx 14 * 3 + 19 * 450 \\
&\approx 8592s \\
&\approx 2.5h
\end{aligned}
$$



Adaptive Linear Relaxation (ALR)

How long to optimize Linux qspinlock?

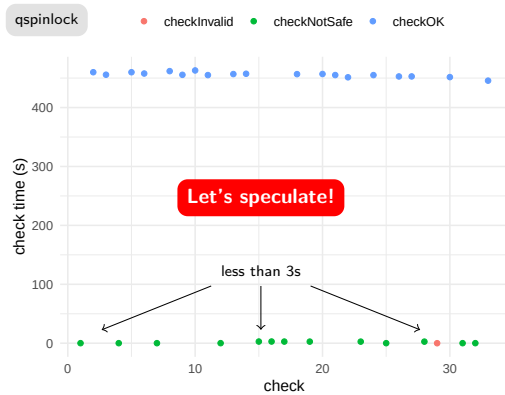
$$
\begin{aligned}
T_o &\approx C_{\text{✗}} * t_{\text{✗}} + C_{\text{✓}} * t_{\text{✓}} \\
&\approx 14 * t_{\text{✗}} + 19 * t_{\text{✓}} \\
&\approx 14 * 3 + 19 * 450 \\
&\approx 8592s \\
&\approx 2.5h
\end{aligned}
$$



Adaptive Linear Relaxation (ALR)

How long to optimize Linux qspinlock?

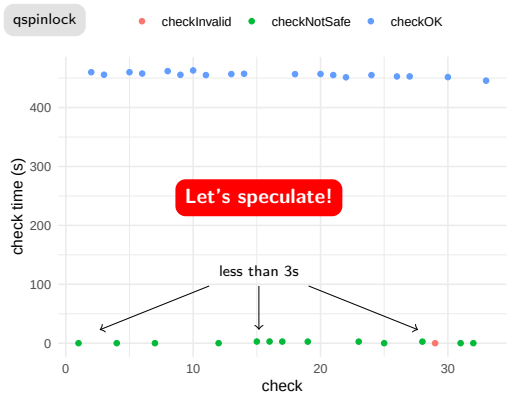
$$
\begin{aligned}
T_o &\approx C_{\lightning} * t_{\lightning} + C_{\checkmark} * t_{\checkmark} \\
&\approx 14 * t_{\lightning} + 19 * t_{\checkmark} \\
&\approx 14 * 3 + 19 * 450 \\
&\approx 8592s \\
&\approx 2.5h
\end{aligned}
$$



Speculation insight: exploring all executions of a *correct* barrier combination typically takes *much longer than* finding one counter-example of an *incorrect* combination.

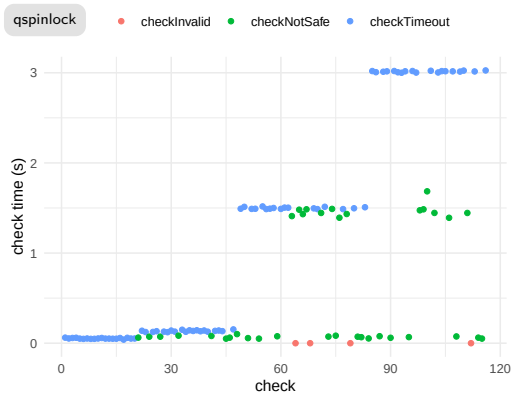
Adaptive Linear Relaxation (ALR)

```
function ALR(b = [sc, ..., sc])
    τ ← 10ms
    while ⊤ do
        b ← LR^τ()
        if b = [sc, ..., sc] ∨ check(b) then
            return b
        τ ← τ + time taken to check b
```

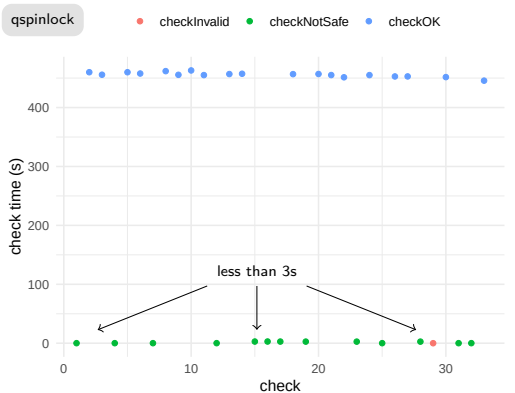


Speculation insight: exploring all executions of a *correct* barrier combination typically takes *much longer than* finding one counter-example of an *incorrect* combination.

Adaptive Linear Relaxation (ALR)

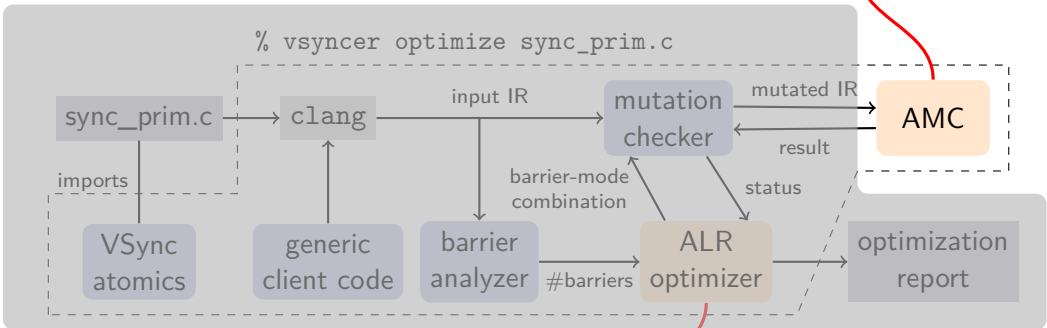


Adaptive Linear Relaxation = 10min



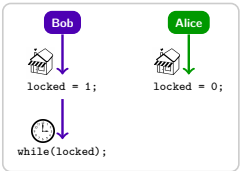
Linear Relaxation = 2.5h

VSync Framework



Stateless Model Checking

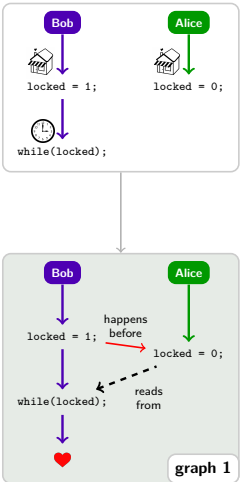
The non-termination problem



- Set of execution graphs = state of *stateless* MCs
- Every *read* has to read from some *write*
- For each graph, construct new graphs to explore

Stateless Model Checking

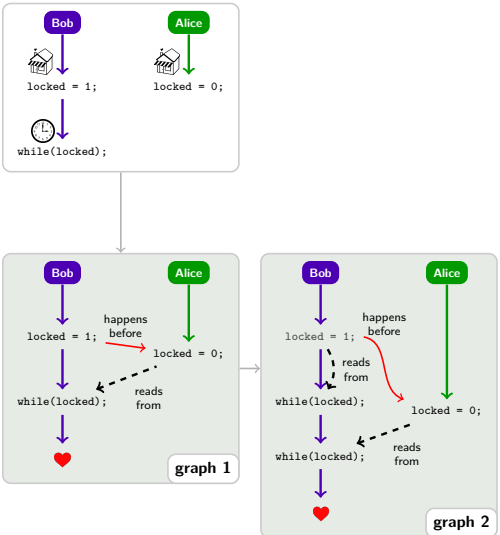
The non-termination problem



- Set of execution graphs = state of *stateless* MCs
- Every *read* has to read from some *write*
- For each graph, construct new graphs to explore

Stateless Model Checking

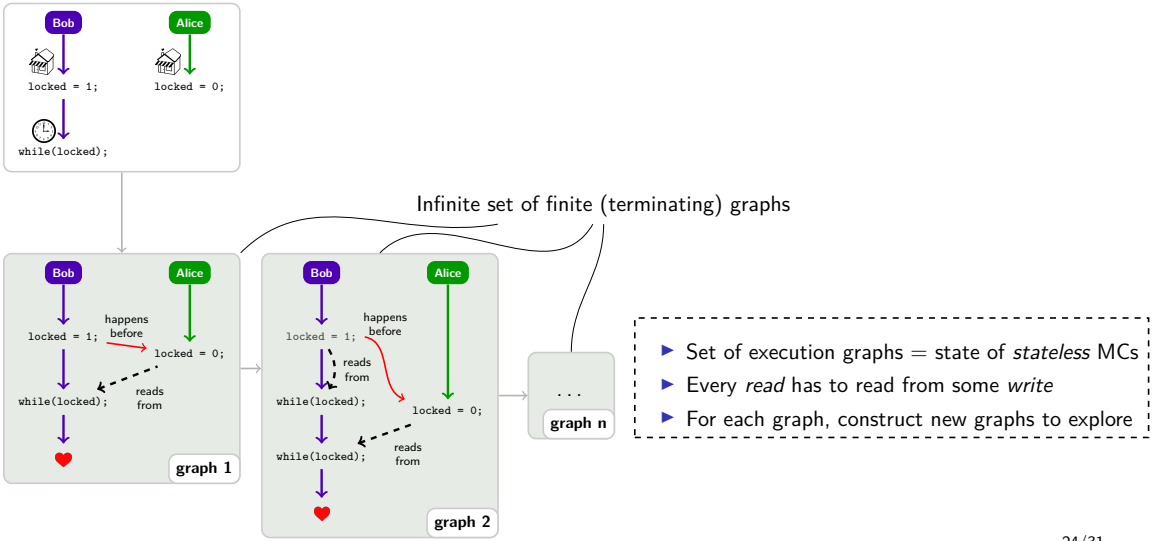
The non-termination problem



- Set of execution graphs = state of *stateless* MCs
- Every *read* has to read from some *write*
- For each graph, construct new graphs to explore

Stateless Model Checking

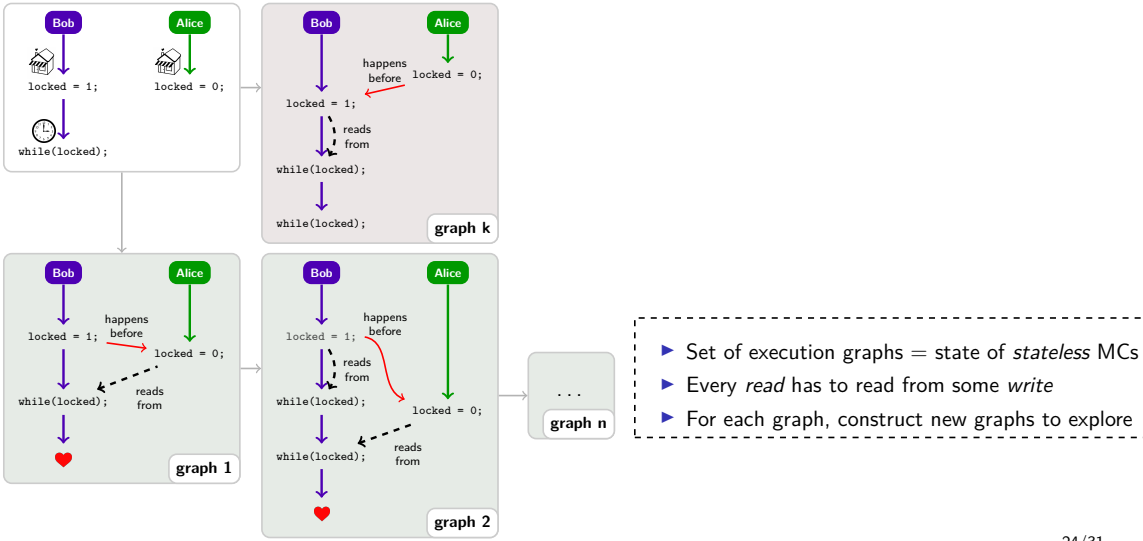
The non-termination problem



- Set of execution graphs = state of *stateless* MCs
- Every *read* has to read from some *write*
- For each graph, construct new graphs to explore

Stateless Model Checking

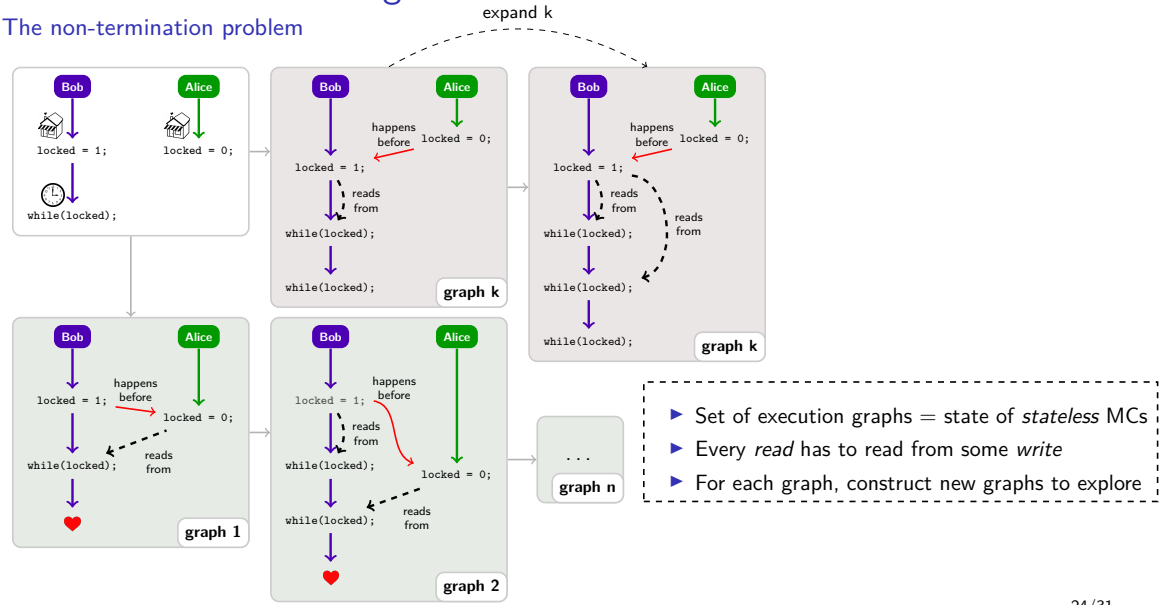
The non-termination problem



- Set of execution graphs = state of *stateless* MCs
- Every *read* has to read from some *write*
- For each graph, construct new graphs to explore

Stateless Model Checking

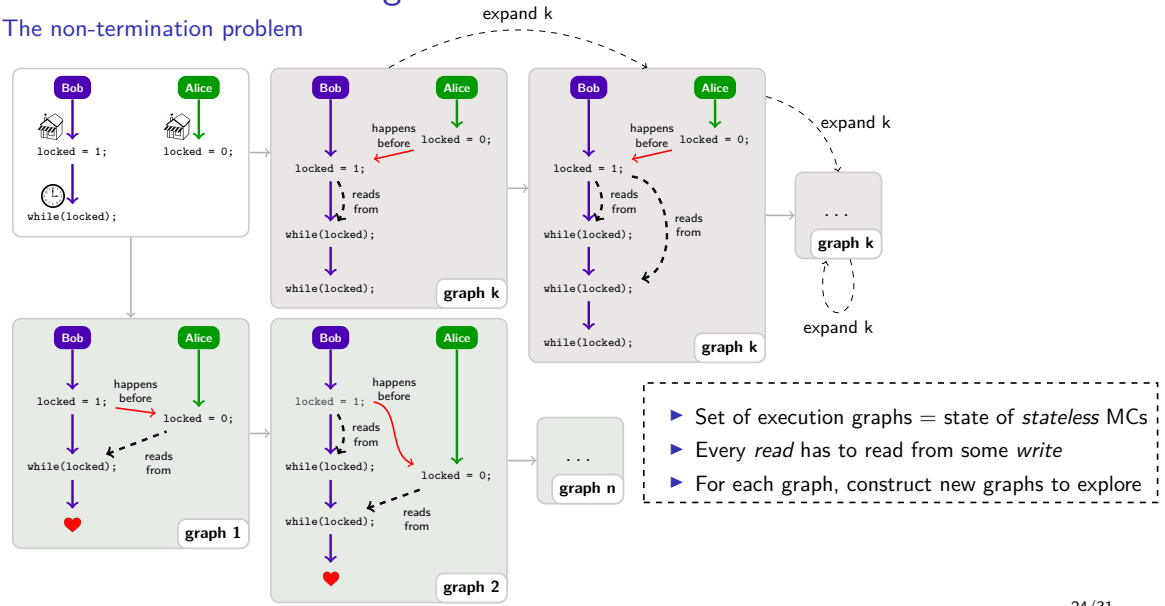
The non-termination problem



- Set of execution graphs = state of *stateless* MCs
- Every *read* has to read from some *write*
- For each graph, construct new graphs to explore

Stateless Model Checking

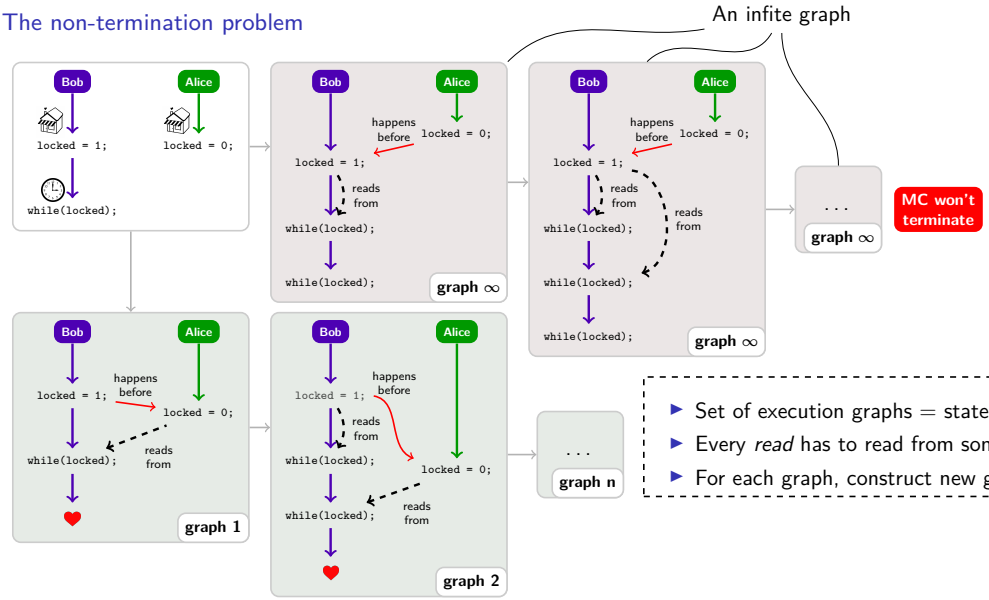
The non-termination problem



- Set of execution graphs = state of *stateless* MCs
- Every *read* has to read from some *write*
- For each graph, construct new graphs to explore

Stateless Model Checking

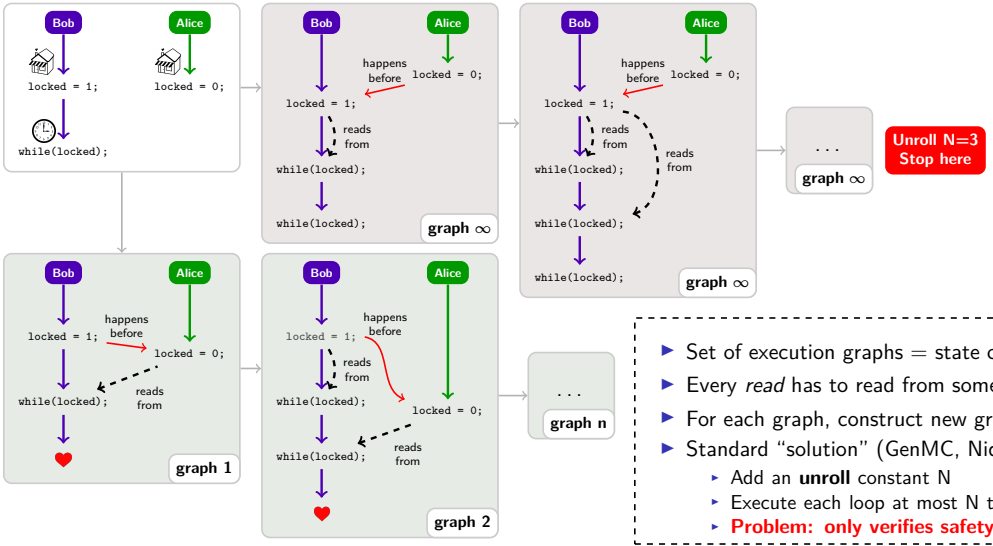
The non-termination problem



- Set of execution graphs = state of *stateless* MCs
- Every *read* has to read from some *write*
- For each graph, construct new graphs to explore

Stateless Model Checking

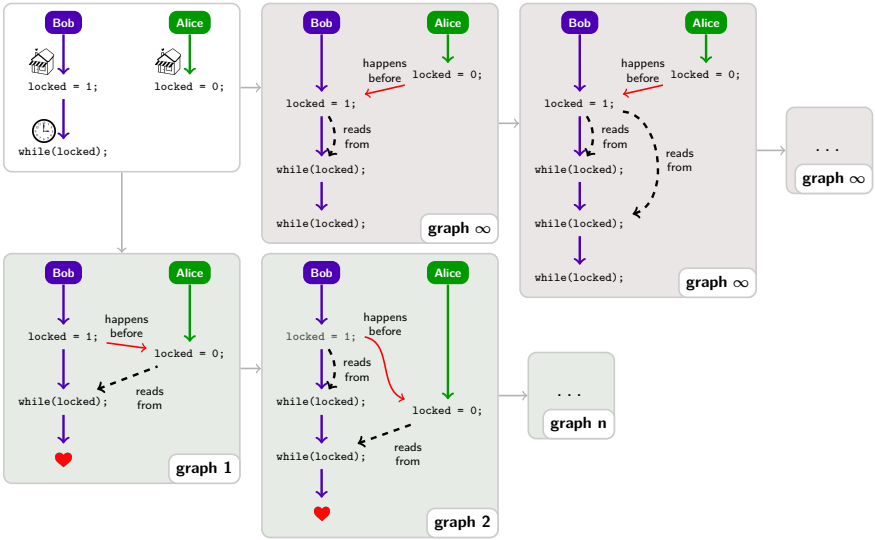
The non-termination problem



- Set of execution graphs = state of *stateless* MCs
- Every *read* has to read from some *write*
- For each graph, construct new graphs to explore
- Standard "solution" (GenMC, Nidhugg, . . .)
  - Add an **unroll** constant N
  - Execute each loop at most N times
  - **Problem: only verifies safety**

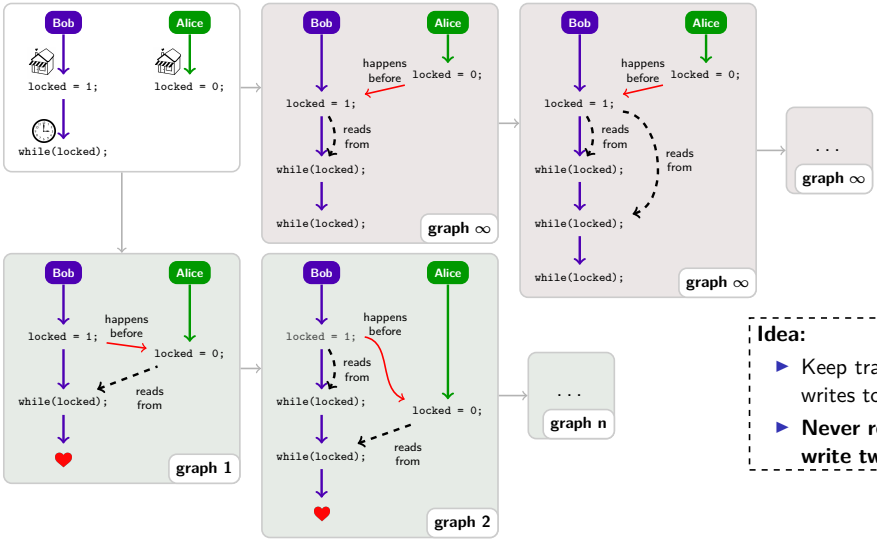
Await Model Checking (AMC)

Detecting non-termination



Await Model Checking (AMC)

Detecting non-termination

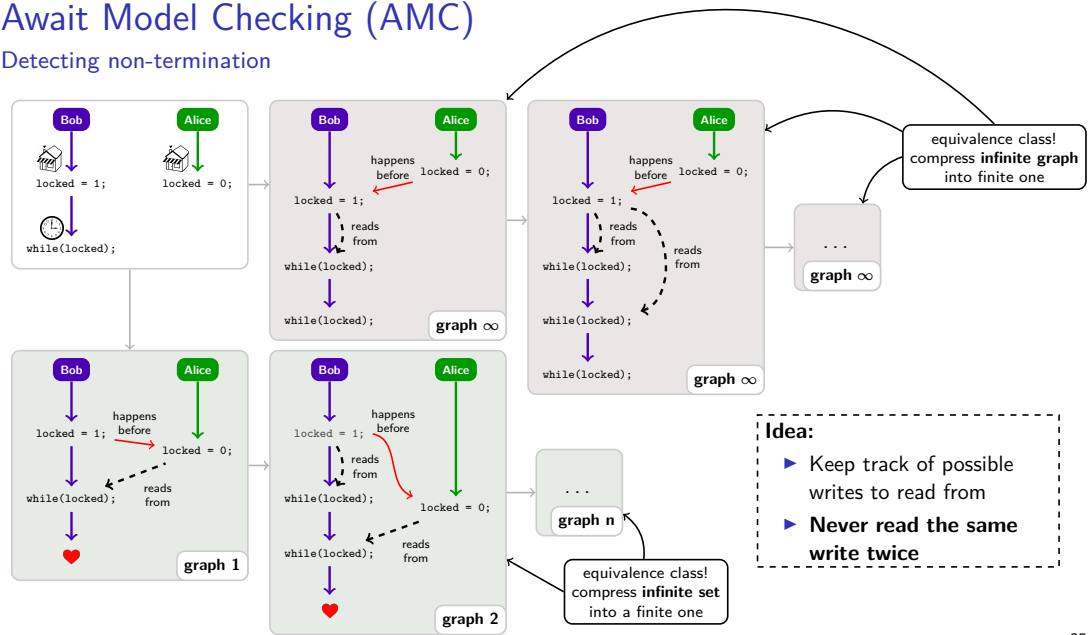


Idea:

- Keep track of possible writes to read from
- **Never read the same write twice**

Await Model Checking (AMC)

Detecting non-termination

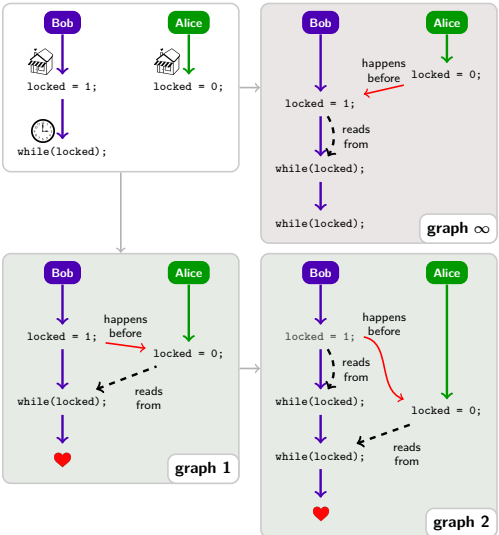


Idea:

- Keep track of possible writes to read from
- **Never read the same write twice**

Await Model Checking (AMC)

Detecting non-termination

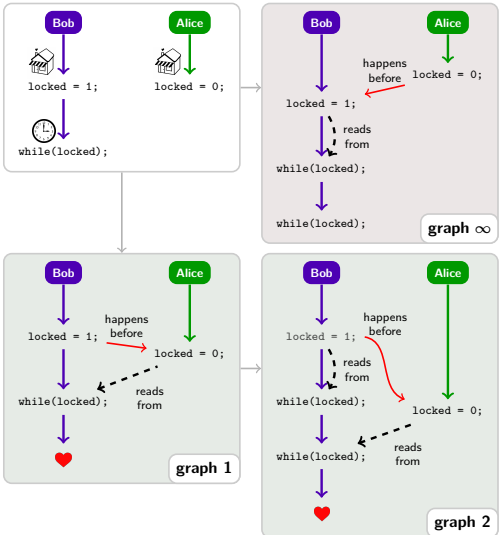


Idea:

- Keep track of possible writes to read from
- **Never read the same write twice**

Await Model Checking (AMC)

Detecting non-termination



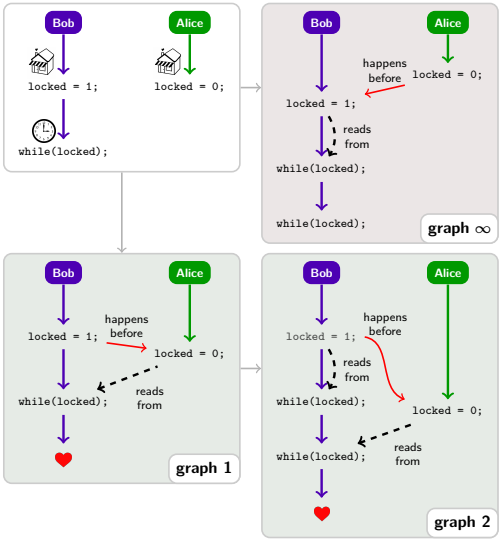
Optimal "unroll":
N at most 1

Idea:

- Keep track of possible writes to read from
- **Never read the same write twice**

Await Model Checking (AMC)

Detecting non-termination



Detect liveness violation
if no writes left
to read from!

Optimal "unroll":
N at most 1

Idea:

- Keep track of possible writes to read from
- **Never read the same write twice**

Await Model Checking (AMC)

Summary

Idea

- Track possible writes to read from
- In an **await loop**, never read the same write twice
- Detect **liveness violation** when
  - All runnable threads are in await loops;
  - And there is no write to read from

Await Model Checking (AMC)

Summary

Idea

- Track possible writes to read from
- In an **await loop**, never read the same write twice
- Detect **liveness violation** when
  - All runnable threads are in await loops;
  - And there is no write to read from

Conditions

- Await loops are side-effect free:
  Except for the last iteration,
  the loop does not modify variables
  eg, `while(!flag);`
- Other loops must be bounded,
  eg, `k = 10; while(k--);`

Agenda

- Motivation
- Challenges
- VSync Framework
  - Adaptive Linear Relaxation (ALR)
  - Await Model Checking (AMC)
- **Evaluation**
- Summary and Outlook

Evaluation

Synchronization Primitive	Number of barriers	Requires AT detection?	Verification time (s)		Optimization time (s)		Optimized-code speedup (%)	
			AMC	GenMC	LR	ALR	x86	ARMv8
CLH lock	4		0.05	0.26	0.40	0.63	732.60	30.92
ArrayQ lock	4		0.05	0.26	0.46	0.60	366.55	38.42
Ticketlock	4		0.12	1.04	0.62	0.60	427.07	6.40
Semaphore	6		0.27	0.83	1.98	1.13	3.22	27.72
CertiKOS MCS	9	✓	1.01	58.15	8.06	2.98	52.52	72.75
TWA lock	10	✓	1.29	12.38	13.78	2.78	268.97	33.53
MCS lock	10	✓	1.54	68.7	15.14	3.01	348.78	73.02
CAS lock	3		6.72	97.96	20.70	7.26	351.32	4.55
RW lock	10		4.57	45.25	38.68	6.19	120.37	43.71
TTAS lock	3		14.72	558.73	45.19	15.35	348.88	7.61
c-TKT-MCS	27	✓	4.86	137.93	126.21	11.21	139.54	66.40
3-state mutex	9	✓	28.22	124.61	159.72	30.10	0.00	3.37
rec. CAS lock	4		44.49	593.81	187.83	46.58	505.49	29.15
c-MCS-TWA	31	✓	14.48	321.94	430.83	63.74	124.45	61.04
c-TTAS-MCS	26	✓	28.68	943.49	762.48	35.22	138.43	54.60
HCLH lock	12		341.22	9986.13	1873.00	456.19	143.25	31.89
qspinlock	26	✓	490.92	17159.73	8678.95	657.74	405.39	51.16
musl mutex	14	✓	11831.69	∞	73536.89	13572.87	20.18	5.00

Evaluation

Synchronization Primitive	Number of barriers	Requires AT detection?	Verification time (s) AMC	Verification time (s) GenMC	Optimization time (s) LR	Optimization time (s) ALR	Optimized-code speedup (%) x86	Optimized-code speedup (%) ARMv8
CLH lock	4		0.05	0.26	0.40	0.63	732.60	30.92
ArrayQ lock	4		0.05	0.26	0.46	0.60	366.55	38.42
Ticketlock	4		0.12	1.04	0.62	0.60	427.07	6.40
Semaphore	6		0.27	0.83	1.98	1.13	3.22	27.72
CertiKOS MCS	9	✓	1.01	58.15	8.06	2.98	52.52	72.75
TWA lock	10	✓	1.29	12.38	13.78	2.78	268.97	33.53
MCS lock	10	✓	1.54	68.7	15.14	3.01	348.78	73.02
CAS lock	3		6.72	97.96				4.55
RW lock	10		4.57	45.25				43.71
TTAS lock	3		14.72	558.73				7.61
c-TKT-MCS	27	✓	4.86	137.93				66.40
3-state mutex	9	✓	28.22	124.61				3.37
rec. CAS lock	4		44.49	593.81	187.83	46.58	505.49	29.15
c-MCS-TWA	31	✓	14.48	321.94	430.83	63.74	124.45	61.04
c-TTAS-MCS	26	✓	28.68	943.49	762.48	35.22	138.43	54.60
HCLH lock	12		341.22	9986.13	1873.00	456.19	143.25	31.89
qspinlock	26	✓	490.92	17159.73	8678.95	657.74	405.39	51.16
musl mutex	14	✓	11831.69	∞	73536.89	13572.87	20.18	5.00

GenMC performance depends on unroll N. Minimum N=4 in these experiments.

Evaluation

Synchronization Primitive	Number of barriers	Requires AT detection?	Verification time (s) AMC	Verification time (s) GenMC	Optimization time (s) LR	Optimization time (s) ALR	Optimized-code speedup (%) x86	Optimized-code speedup (%) ARMv8
CLH lock	4		0.05	0.26	0.40	0.63	732.60	30.92
ArrayQ lock	4		0.05	0.26	0.46	0.60	366.55	38.42
Ticketlock	4		0.12	1.04	0.62	0.60	427.07	6.40
Semaphore	6		0.27	0.83	1.98	1.13	3.22	27.72
CertiKOS MCS	9	✓	1.01	58.15	8.06	2.98	52.52	72.75
TWA lock	10	✓	1.29	12.38	13.78	2.78	268.97	33.53
MCS lock	10	✓	1.54	68.7	15.14	3.01	348.78	73.02
CAS lock	3		6.72	97.96	20.70	7.26	351.32	4.55
RW lock	10		4.57	45.25	38.68	6.19	120.37	43.71
TTAS lock	3				.19	15.35	348.88	7.61
c-TKT-MCS	27	✓			.21	11.21	139.54	66.40
3-state mutex	9	✓			.72	30.10	0.00	3.37
rec. CAS lock	4				.83	46.58	505.49	29.15
c-MCS-TWA	31	✓	14.48	321.94	430.83	63.74	124.45	61.04
c-TTAS-MCS	26	✓	28.68	943.49	762.48	35.22	138.43	54.60
HCLH lock	12		341.22	9986.13	1873.00	456.19	143.25	31.89
qspinlock	26	✓	490.92	17159.73	8678.95	657.74	405.39	51.16
musl mutex	14	✓	11831.69	∞	73536.89	13572.87	20.18	5.00

Most primitives may hang with too weak barriers. So, AMC is *essential*!

Evaluation

Synchronization Primitive	Number of barriers	Requires AT detection?	Verification time (s) AMC	Verification time (s) GenMC	Optimization time (s) LR	Optimization time (s) ALR	Optimized-code speedup (%) x86	Optimized-code speedup (%) ARMv8
CLH lock	4		0.05	0.26	0.40	0.63	732.60	30.92
ArrayQ lock	4		0.05	0.26	0.46	0.60	366.55	38.42
Ticketlock	4		0.12	1.04	0.62	0.60	427.07	6.40
Semaphore	6		0.27	0.83	1.98	1.13	3.22	27.72
CertiKOS MCS	9	✓	1.01	58.15	8.06	2.98	52.52	72.75
TWA lock	10	✓	1.29	12.38	13.78	2.78	268.97	33.53
MCS lock	10	✓	1.54	68.7	15.14	3.01	348.78	73.02
CAS lock	3		6.72	97.96	20.70	7.26	351.32	4.55
RW lock	10			5	38.68	6.19	120.37	43.71
TTAS lock	3			3	45.19	15.35	348.88	7.61
c-TKT-MCS	27			3	126.21	11.21	139.54	66.40
3-state mutex	9	✓	28.22	124.61	159.72	30.10	0.00	3.37
rec. CAS lock	4		44.49	593.81	187.83	46.58	505.49	29.15
c-MCS-TWA	31	✓	14.48	321.94	430.83	63.74	124.45	61.04
c-TTAS-MCS	26	✓	28.68	943.49	762.48	35.22	138.43	54.60
HCLH lock	12		341.22	9986.13	1873.00	456.19	143.25	31.89
qspinlock	26	✓	490.92	17159.73	8678.95	657.74	405.39	51.16
musl mutex	14	✓	11831.69	∞	73536.89	13572.87	20.18	5.00

Search space from 4^4 up to 4^{31} barrier combinations

Evaluation

Synchronization Primitive	Number of barriers	Requires AT detection?	Verification time (s) AMC	Verification time (s) GenMC	Optimization time (s) LR	Optimization time (s) ALR	Optimized-code speedup (%) x86	Optimized-code speedup (%) ARMv8
CLH lock	4		0.05	0.26	0.40	0.63	732.60	30.92
ArrayQ lock	4		0.05	0.26	0.46	0.60	366.55	38.42
Ticketlock	4		0.12	1.04	0.62	0.60	427.07	6.40
Semaphore	6		0.27	0.83	1.98	1.13	3.22	27.72
CertiKOS MCS	9	✓	1.01	58.15	8.06	2.98	52.52	72.75
TWA lock	10	✓	1.29	12.38	13.78	2.78	268.97	33.53
MCS lock	10	✓	1.54	68.7	15.14	3.01	348.78	73.02
CAS lock	3		6.72	97.96	20.70	7.26	351.32	4.55
RW lock	10		4.57	45.25	38.68	6.19	120.37	43.71
TTAS lock	3		4.57	3	45.19	15.35	348.88	7.61
c-TKT-MCS	27		44.49	3	126.21	11.21	139.54	66.40
3-state mutex	9		44.49	1	159.72	30.10	0.00	3.37
rec. CAS lock	4		44.49	593.81	187.83	46.58	505.49	29.15
c-MCS-TWA	31	✓	14.48	321.94	430.83	63.74	124.45	61.04
c-TTAS-MCS	26	✓	28.68	943.49	762.48	35.22	138.43	54.60
HCLH lock	12		341.22	9986.13	1873.00	456.19	143.25	31.89
qspinlock	26	✓	490.92	17159.73	8678.95	657.74	405.39	51.16
musl mutex	14	✓	11831.69	∞	73536.89	13572.87	20.18	5.00

For most cases, ALR is 3% to 95% faster than LR

Evaluation

Synchronization Primitive	Number of barriers	Requires AT detection?	Verification time (s) AMC	Verification time (s) GenMC	Optimization time (s) LR	Optimization time (s) ALR	Optimized-code speedup (%) x86	Optimized-code speedup (%) ARMv8
CLH lock	4		0.05	0.26	0.40	0.63	732.60	30.92
ArrayQ lock	4		0.05	0.26	0.46	0.60	366.55	38.42
Ticketlock	4		0.12	1.04	0.62	0.60	427.07	6.40
Semaphore	6		0.27	0.83	1.98	1.13	3.22	27.72
CertiKOS MCS	9	✓	1.01	58.15	8.06	2.98	52.52	72.75
TWA lock	10	✓	1.29	12.38	13.78	2.78	268.97	33.53
MCS lock	10	✓	1.54	68.7	15.14	3.01	348.78	73.02
CAS lock	3		6.72	97.06	20.70	7.26	351.32	4.55
RW lock	10					6.19	120.37	43.71
TTAS lock	3					6.35	348.88	7.61
c-TKT-MCS	27	✓				1.21	139.54	66.40
3-state mutex	9	✓	28.22	124.61	159.72	30.10	0.00	3.37
rec. CAS lock	4		44.49	593.81	187.83	46.58	505.49	29.15
c-MCS-TWA	31	✓	14.48	321.94	430.83	63.74	124.45	61.04
c-TTAS-MCS	26	✓	28.68	943.49	762.48	35.22	138.43	54.60
HCLH lock	12		341.22	9986.13	1873.00	456.19	143.25	31.89
qspinlock	26	✓	490.92	17159.73	8678.95	657.74	405.39	51.16
musl mutex	14	✓	11831.69	∞	73536.89	13572.87	20.18	5.00

Maximally-relaxed primitives can be much faster than their SC-only counterparts.

Case study: Linux qspinlock

Version	*acq*	*rel*	*sc*	Time	Correctness
Linux 4.4	3	6	6	2015/09/11	Not verified
Linux 4.5	6	2	1	2015/11/09	Barrier bug (fixed in 4.16)
Linux 4.8	6	3	0	2016/06/03	Barrier bug (fixed in 4.16)
Linux 4.16	6	4	0	2018/02/13	Not verified
Linux 5.6	6	2	1	2020/01/07	Not verified
VSync	7	2	1	11 minutes	VSync-verified

Optimization comparison

- VSync-version based on 4.4

Case study: Linux qspinlock

Version	*acq*	*rel*	*sc*	Time	Correctness
Linux 4.4	3	6	6	2015/09/11	Not verified
Linux 4.5	6	2	1	2015/11/09	Barrier bug (fixed in 4.16)
Linux 4.8	6	3	0	2016/06/03	Barrier bug (fixed in 4.16)
Linux 4.16	6	4	0	2018/02/13	Not verified
Linux 5.6	6	2	1	2020/01/07	Not verified
VSync	7	2	1	11 minutes	VSync-verified

Optimization comparison

- VSync-version based on 4.4
- Equivalent optimization, +1 barrier
- 11 minutes to optimize with ALR
- Experts had a harder time

Case study: Linux qspinlock

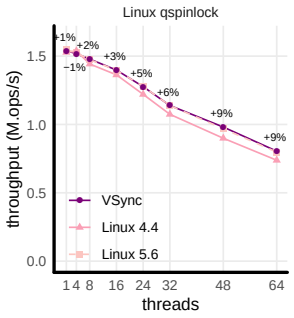
Version	*acq*	*rel*	*sc*	Time	Correctness
Linux 4.4	3	6	6	2015/09/11	Not verified
Linux 4.5	6	2	1	2015/11/09	Barrier bug (fixed in 4.16)
Linux 4.8	6	3	0	2016/06/03	Barrier bug (fixed in 4.16)
Linux 4.16	6	4	0	2018/02/13	Not verified
Linux 5.6	6	2	1	2020/01/07	Not verified
VSync	7	2	1	11 minutes	VSync-verified

Optimization comparison

- VSync-version based on 4.4
- Equivalent optimization, +1 barrier
- 11 minutes to optimize with ALR
- Experts had a harder time

Case study: Linux qspinlock

Version	*acq*	*rel*	*sc*	Time	Correctness
Linux 4.4	3	6	6	2015/09/11	Not verified
Linux 4.5	6	2	1	2015/11/09	Barrier bug (fixed in 4.16)
Linux 4.8	6	3	0	2016/06/03	Barrier bug (fixed in 4.16)
Linux 4.16	6	4	0	2018/02/13	Not verified
Linux 5.6	6	2	1	2020/01/07	Not verified
VSync	7	2	1	11 minutes	VSync-verified



Optimization comparison

- VSync-version based on 4.4
- Equivalent optimization, +1 barrier
- 11 minutes to optimize with ALR
- Experts had a harder time

Performance comparison

- Run on Kunpeng 920 server (AArch64), Kyoto Cabinet DB
- VSync-version has equivalent performance to 5.6: $< 1\%$

Agenda

- Motivation
- Challenges
- VSync Framework
  - Adaptive Linear Relaxation (ALR)
  - Await Model Checking (AMC)
- Evaluation
- **Summary and Outlook**

Summary and Outlook (We are hiring!)

VSync-framework

- **Adaptive Linear Relaxtion (ALR)**
  - traverses the search space in linear number of steps
  - caps model checker runs with an adaptive timeout.
- **Await Model Checking (AMC)**
  - detects non-terminating await loops on WMMs for the first time
  - suffices for verifying liveness of synchronization primitives
- **A library of atomic operations**

Synchronization primitives

- Verified and optimized **more than 15** primitives, **many for the first time**
- DPDK MCS bug reported and fixed
- seL4 CLH bug reported and fixed
- VSync-optimize primitive perform similar as expert-optimized,eg, Linux qspinlock

Future work

- Verifying linearizability and progress properties for concurrent objects on WMMs