OFence: Pairing Barriers to Find Concurrency Bugs in the Linux Kernel

Baptiste Lepers
Josselin Giet
Julia Lawall
Willy Zwaenepoel

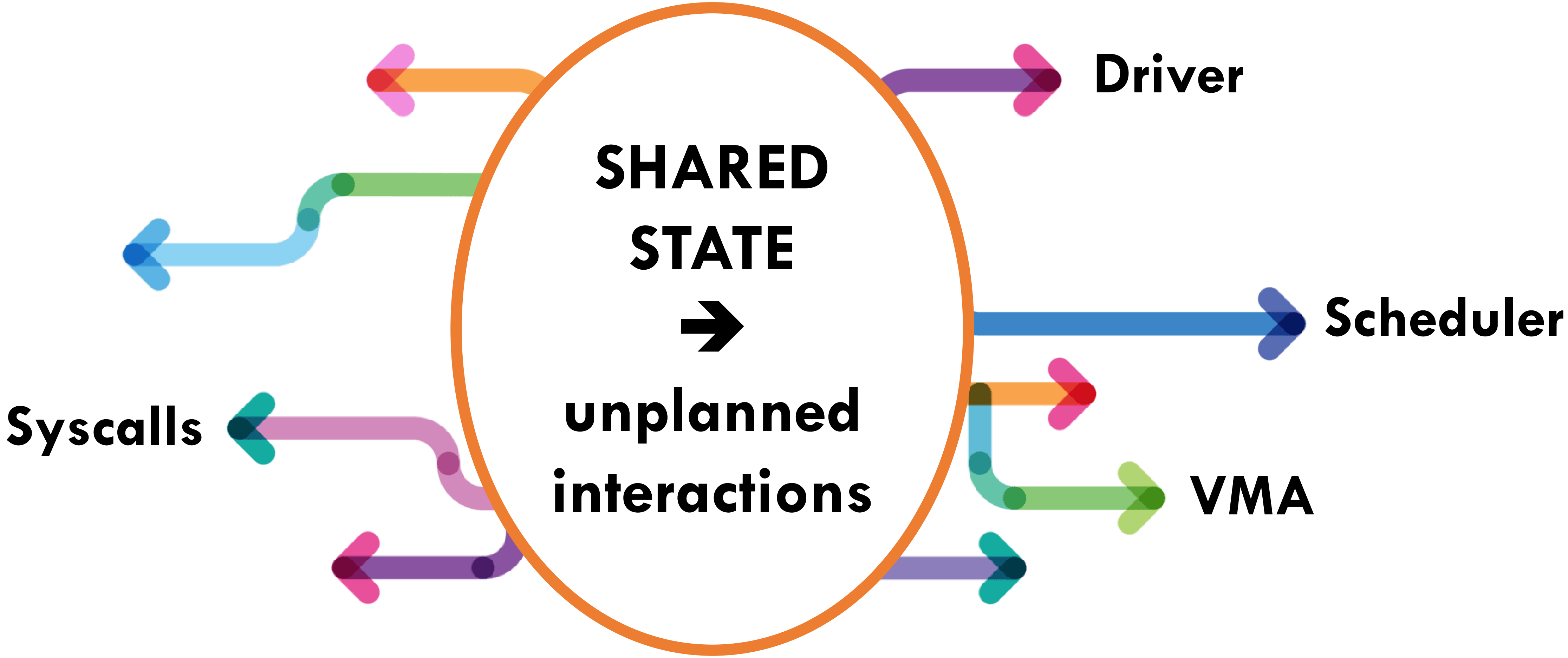
THE UNIVERSITY OF SYDNEY

Inria

Linux is a highly concurrent system

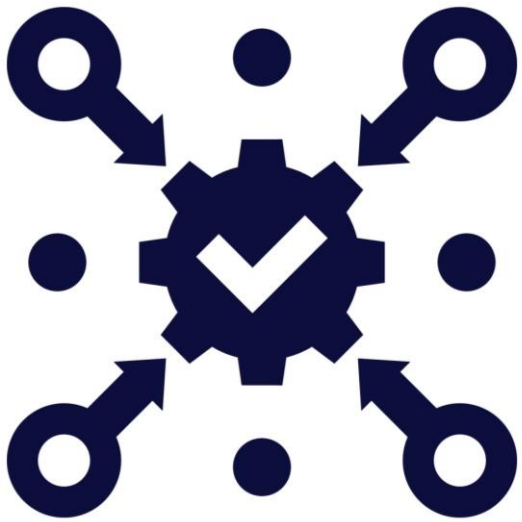
2/58

Linux is a highly concurrent system



3/58

How to find concurrency bugs?



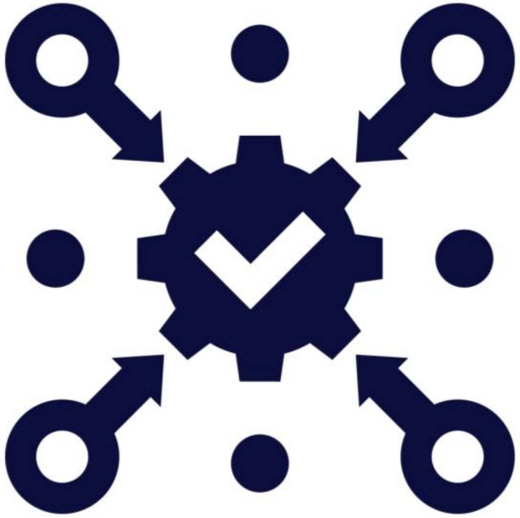
1. **Dynamic analysis (test & observe)**

    Race reported on correct code

    Limited coverage

4/58

How to find concurrency bugs?



1. Dynamic analysis (test & observe)
   - *Race reported on correct code*
   - *Limited coverage*

2. Static analysis)
   - *Limited to lock-based code*

5/58

Limitation of the state-of-the-art

```
lock(&identifier)
shared_var = xxx;
unlock(&identifier)
```

```
lock(&identifier)
…
unlock(&identifier)
if(shared_var) {…}
```

Traditional approach: locksets.

If 2 functions use the same lock
Then they likely run concurrently

If a variable is modified in a critical section and read outside → bug

Lockless code not analyzed.

6/58

This talk

Analyze concurrent lockless code.

First step: infer concurrency.

7/58

How to analyze lockless code?

Observation 1: lockless code (often) uses barriers.

```
1.      write a
2.      mfence
3.      write b
```

CPU *must* write a before writing b

8/58

Idea: use barriers

Problem: barriers do not have an identifier

```
1.      write a
2.      mfence
3.      write b
```


9/58

Idea: pair barriers

Observation 2: barriers run in pairs.

```
struct type *s = …;
s.field = …;
mfence
s.initialized = 1;
```

```
if(!s.initialized)
        return;
mfence
f(s.field);
```


10/58

Idea: pair barriers

Observation 2: barriers run in pairs.

```
struct type *s = …;
s.initialized = 1;
s.field = …;
```

```
if(!s.initialized)
        return;
mfence
f(s.field);
```


11/58

Idea: pair barriers using shared objects

Observation 3: avoid aliasing using types

```
1.    struct type *s = …;
2.    s.field = …;
3.    mfence
4.    s.init = 1;
```

```
1.    if(!x.initialized)
2.        return;
3.    mfence
4.    f(x.field);
```

```
2.    write (type, field)
3.    mfence
4.    write (type, init)
```

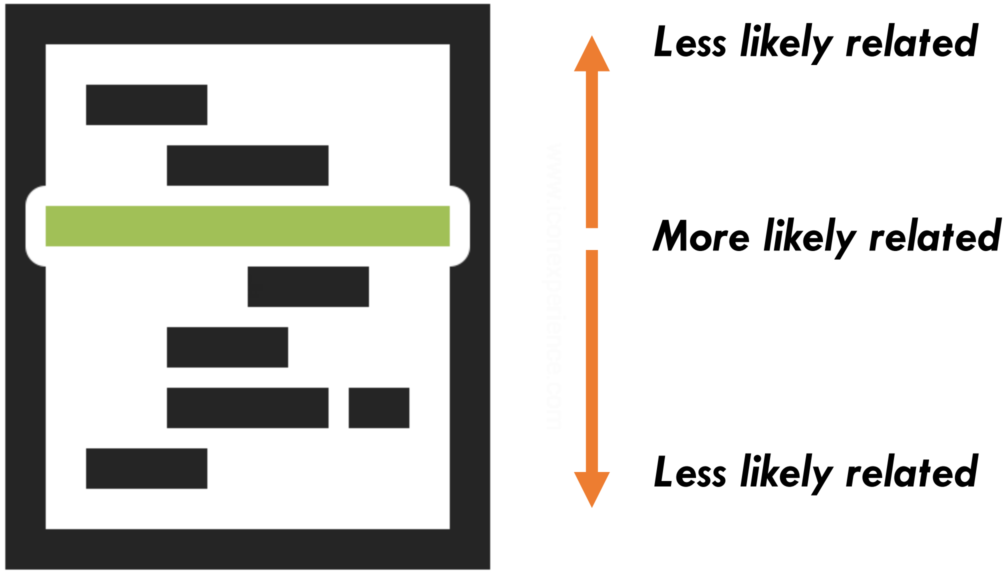
```
1.    read (type, init)
3.    mfence
4.    read (type, field)
```

We call (type of struct, field name) a « shared object ».

12/58

Idea: pair barriers using closest shared objects

Observation 4: code related to a barrier is close to the barrier.

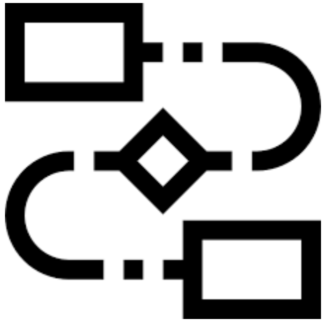


13/58

Implementation

1. Keep files with barriers

Per barrier

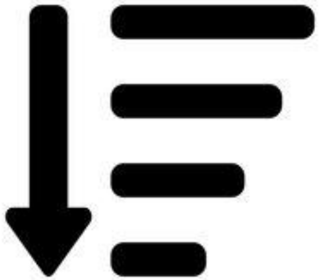


2. Produce Control Flow Graphs

Limit to 10 statements before and 10 statement after each barrier

3. Extract shared objects

Per barrier



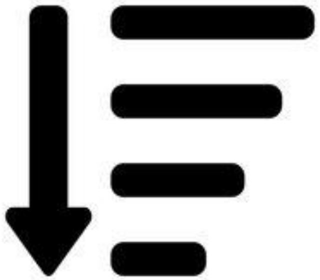
4. Sort shared objects by distance

Distance = number of statements to/from barrier

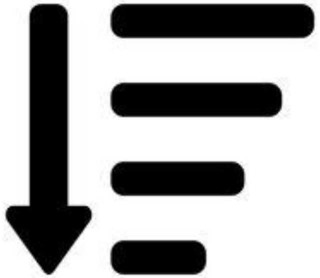
14/58

Implementation: pairing

Barrier 1



Barrier 2



1. *Have 2 shared objects o1 and o2 in common?*
2. *Objects are ordered by at least 1 barrier?*
3. *Pair with weight:*

*weight = o1.distance(barrier1) * o2.distance(barrier1)*
*+ o1.distance(barrier2) * o2.distance(barrier2)*

4. If a barrier is paired with 2 barriers, keep the pairing with lowest weight.

15/58

Use case: check ordering constraints

A barrier is only useful if:

before(barrier 1, a) ⇔ after(barrier 2, a)

```
write a
mfence
write b
```

```
read b
mfence
read a
```

```
mfence
read b
read a
```

	New b	Old b
New a	✓	✓
Old a		✓

16/58

Use case: remove unneeded barriers

If a barrier is unpaired, is it needed?

write a
~~mfence~~
compare_and_swap(…)

write a
~~mfence~~
function_with_barrier_semantics(…)

17/58

Evaluation

Entire Linux code base

12 new bugs found & fixed

Could have led to serious and hard-to-debug crashes

50 uneeded barriers detected

Patches have been merged.

18/58

Misplaced access in the RPC interface

```
void xprt_complete_rqst (...) {
      req->rq_private_buf.len = ....;
      smp_wmb();
      req->rq_reply_bytes_recvd = copied;
}

static void call_decode (...) {
+     if (!req->rq_reply_bytes_recvd)
              goto out;
      smp_rmb();
-     if (!req->rq_reply_bytes_recvd)
              goto out;
      req->rq_rcv_buf.len = req->rq_private_buf.len;
```


19/58

Racy re-read in the socket interface

```
1 int reuseport_add_sock(...) {
2       reuse->socks[reuse->num_socks] = sk ;
3       smp_wmb();
4       reuse->num_socks++;
5 }

469 struct sock *reuseport_select_sock(...) {
470     int socks = READ_ONCE(reuse->num_socks);
471     if (likely(socks)) {
472             smp_rmb();
…
487             reuseport_select_sock_by_hash(reuse);
```

 20/58

Racy re-read in the socket interface

```
int reuseport_add_sock(...) {
      reuse->socks[reuse->num_socks] = sk ;
      smp_wmb();
      reuse->num_socks++;
}

struct sock *reuseport_select_sock(...) {
      int socks = READ_ONCE(reuse->num_socks);
```

```
static struct sock *reuseport_select_sock_by_hash(…) {
…
      while (reuse->socks[i]->sk_state == …) {
            i++;
            if (i >= reuse->num_socks)
                  i = 0;
…           …
…     }
… }
```


21/58

Racy re-read in the socket interface

```
int reuseport_add_sock(...) {
      reuse->socks[reuse->num_socks] = sk ;
      smp_wmb();
      reuse->num_socks++;
}

struct sock *reuseport_select_sock(...) {
      int socks = READ_ONCE(reuse->num_socks);
…
```

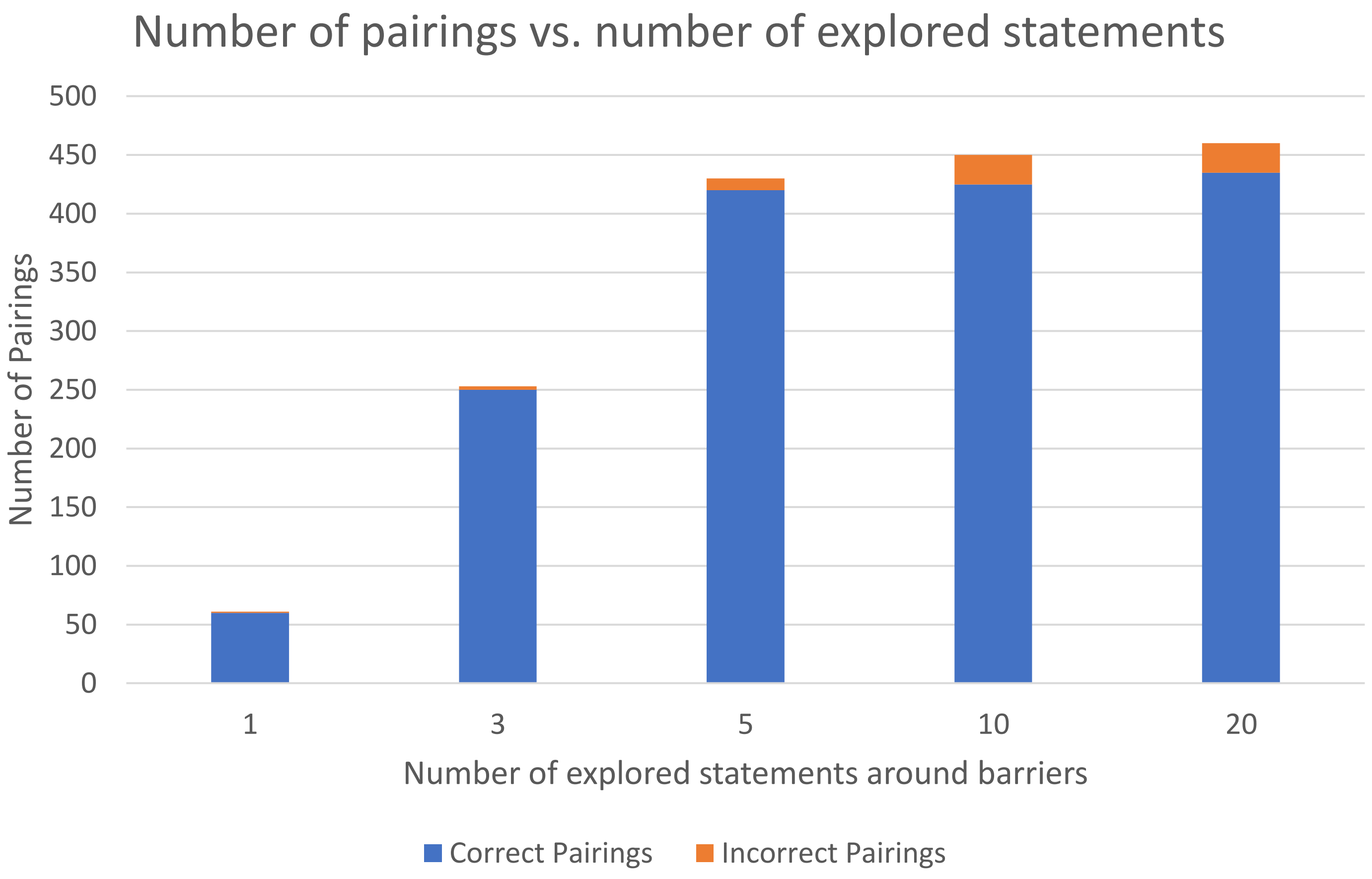
```
static struct sock *reuseport_select_sock_by_hash(…, socks){
…
      while (reuse->socks[i]->sk_state == …) {
            i++;
            if (i >= socks)
                  i = 0;
…           …
      }
…
… }
```

Removing unneeded barrier

```
static int rq_qos_wake_function (...) {
      data->got_token = true ;
-     smp_wmb();
      wake_up_process(data->task);
}
```


23/58

Influence of tuning parameters



24/58

Extension #1: avoid reports on benign races

Dynamic analysers report races on correct code. We mark the code as safe.

```
  static int __pollwake (...) {
        smp_wmb();
-       pwq->triggered = 1;
+       WRITE_ONCE(pwq->triggered, 1);
        return ...;
  }

  static int poll_schedule_timeout (...) {
-       if (!pwq->triggered)
+       if (!READ_ONCE(pwq->triggered))
              rc = schedule_hrtimeout_range (...);
        smp_store_mb(…); // equivalent to smp_mb
  }
```


25/58

Extension #2: avoid load/store tearing

Reads/writes on some shared objects have to be atomic.

```
  void xprt_complete_rqst (...) {
        req->rq_private_buf.len = …;
        smp_wmb();
-       req->rq_reply_bytes_recvd = copied;
+       WRITE_ONCE(req->rq_reply_bytes_recvd,copied);
  }
```

Without WRITE_ONCE, the compiler could do:

write 1st 32b of copied to rq_reply_bytes

write 2nd 32b of copied to rq_reply_bytes

(Actually happens with clang on arm64 CPUs!)

… resulting in readers reading possibly partially written values.

26/58

Conclusion (of the 1st part of the talk)

It is possible to infer concurrency by pairing barriers

- Shared objects
- Distance

Lockless code is error prone

Proof of concept: orderings – 12 bugs

Use it to check other concurrency bugs! (Use-after-free, …)

Questions (before the 2nd part)?

27/58

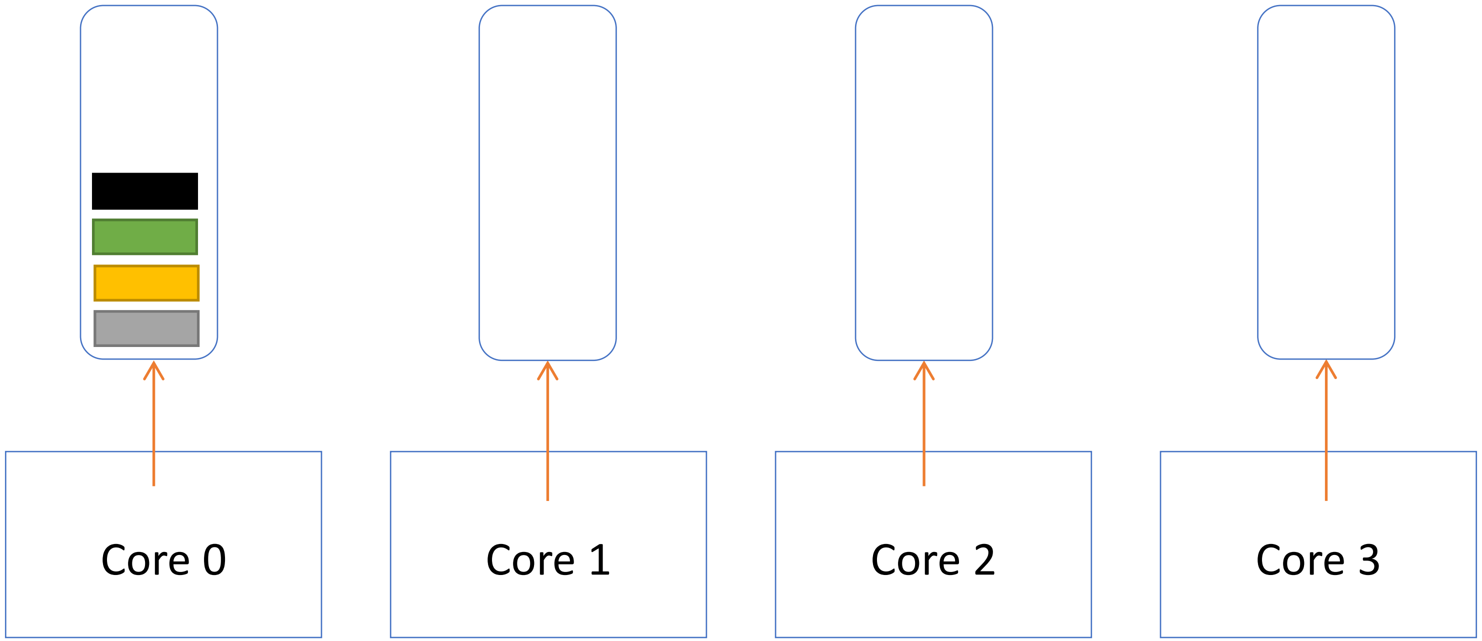
Provable Multicore Schedulers with Ipanema: Application to Work-Conservation

Baptiste Lepers
Redha Gouicem
Damien Carver
Jean-Pierre Lozi
Nicolas Palix
Virginia Aponte
Willy Zwaenepoel
Julien Sopena
Julia Lawall
Gilles Muller

THE UNIVERSITY OF SYDNEY
SORBONNE UNIVERSITÉ
LIP6
UNIVERSITÉ Grenoble Alpes
Inria
cnam
ORACLE

Work conservation

“No core should be left idle when a core is overloaded”

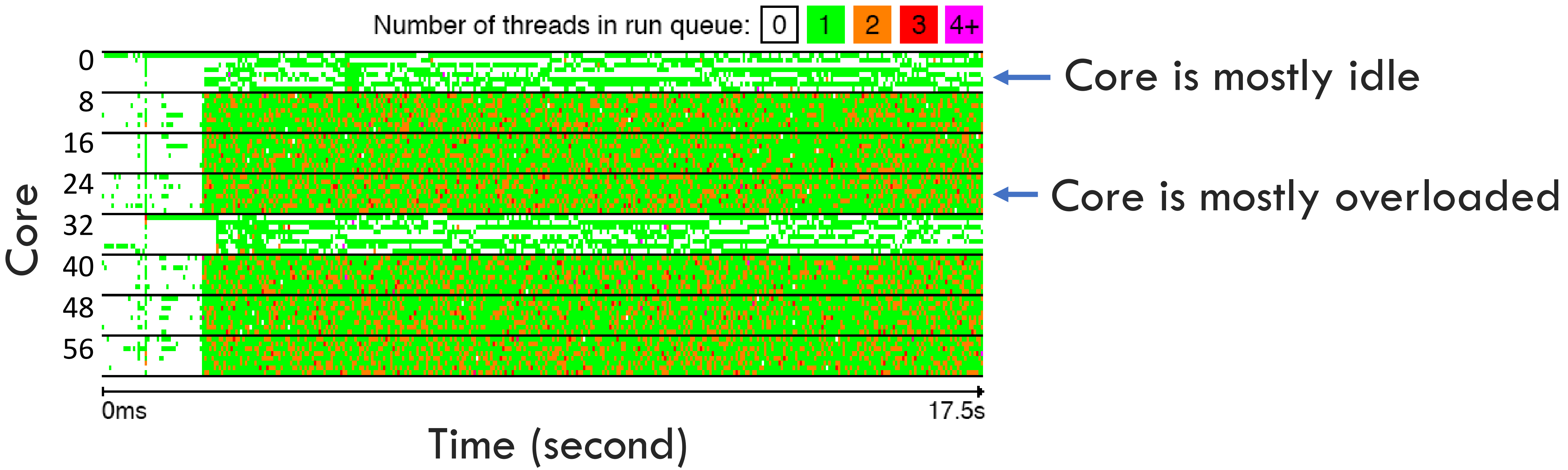


Non work-conserving situation:
core 0 is overloaded, other cores are idle

29/58

Problem

Linux (CFS) suffers from work conservation issues

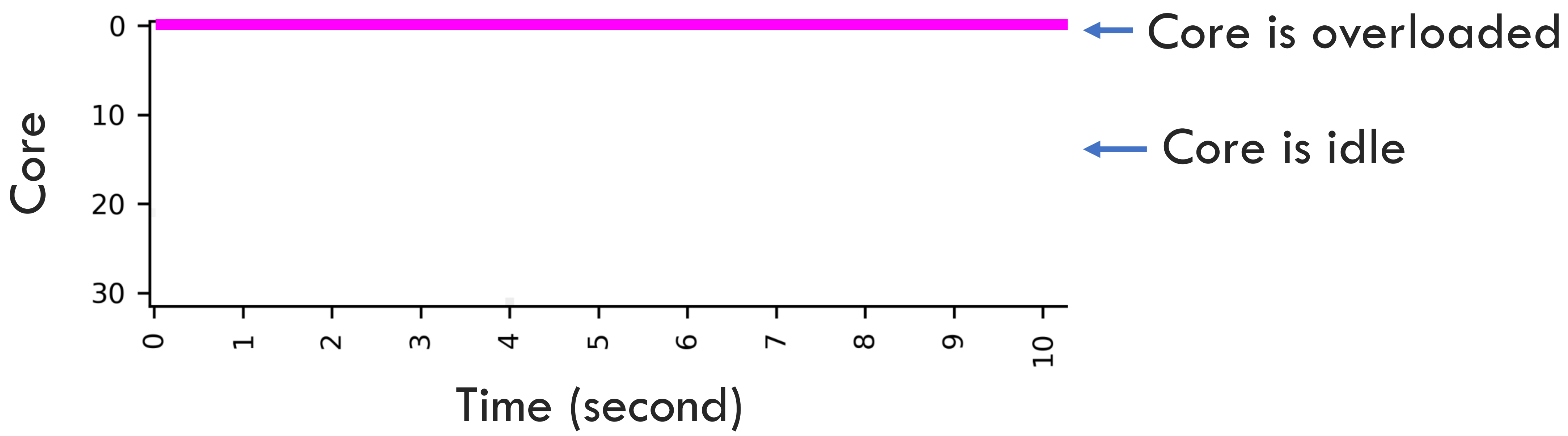


[Lozi et al. 2016]

30/58

Problem

FreeBSD (ULE) suffers from work conservation issues



[Bouron et al. 2018]

31/58

Problem

Work conservation bugs are hard to detect

No crash, no deadlock. No obvious symptom.

137x slowdown on HPC applications
23% slowdown on a database.
[Lozi et al. 2016]

32/58

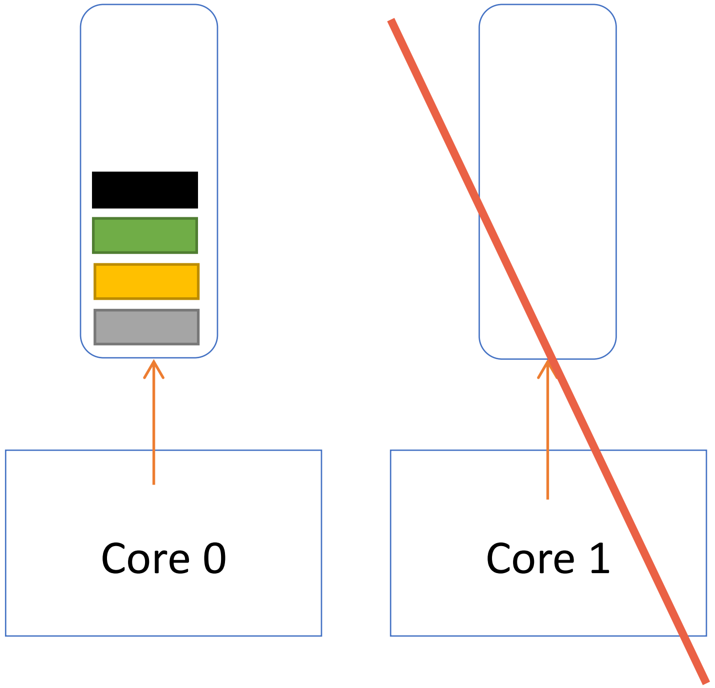
This talk

Formally prove work-conservation

Work Conservation Formally

$$(\exists c\ .\ O(c)) \Rightarrow (\forall c'\ .\ \neg I(c'))$$

If a core is overloaded, no core is idle

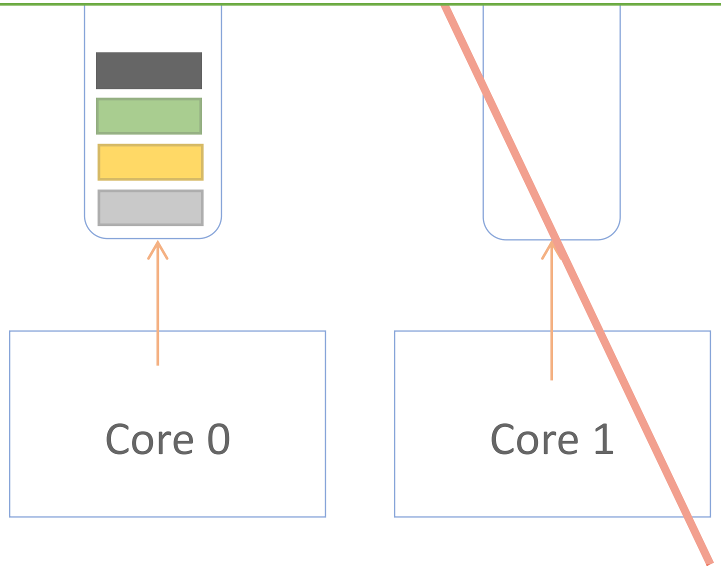


34/58

Work Conservation Formally

$$(\exists c\ .\ O(c)) \Rightarrow (\forall c'\ .\ \neg I(c'))$$

Does not work for realistic schedulers!



35/58

Challenge #1

Concurrent events & optimistic concurrency

36/58

Challenge #1

Concurrent events & optimistic concurrency

time

Observe (state of **every** core)

Lock (**one** core – less overhead)

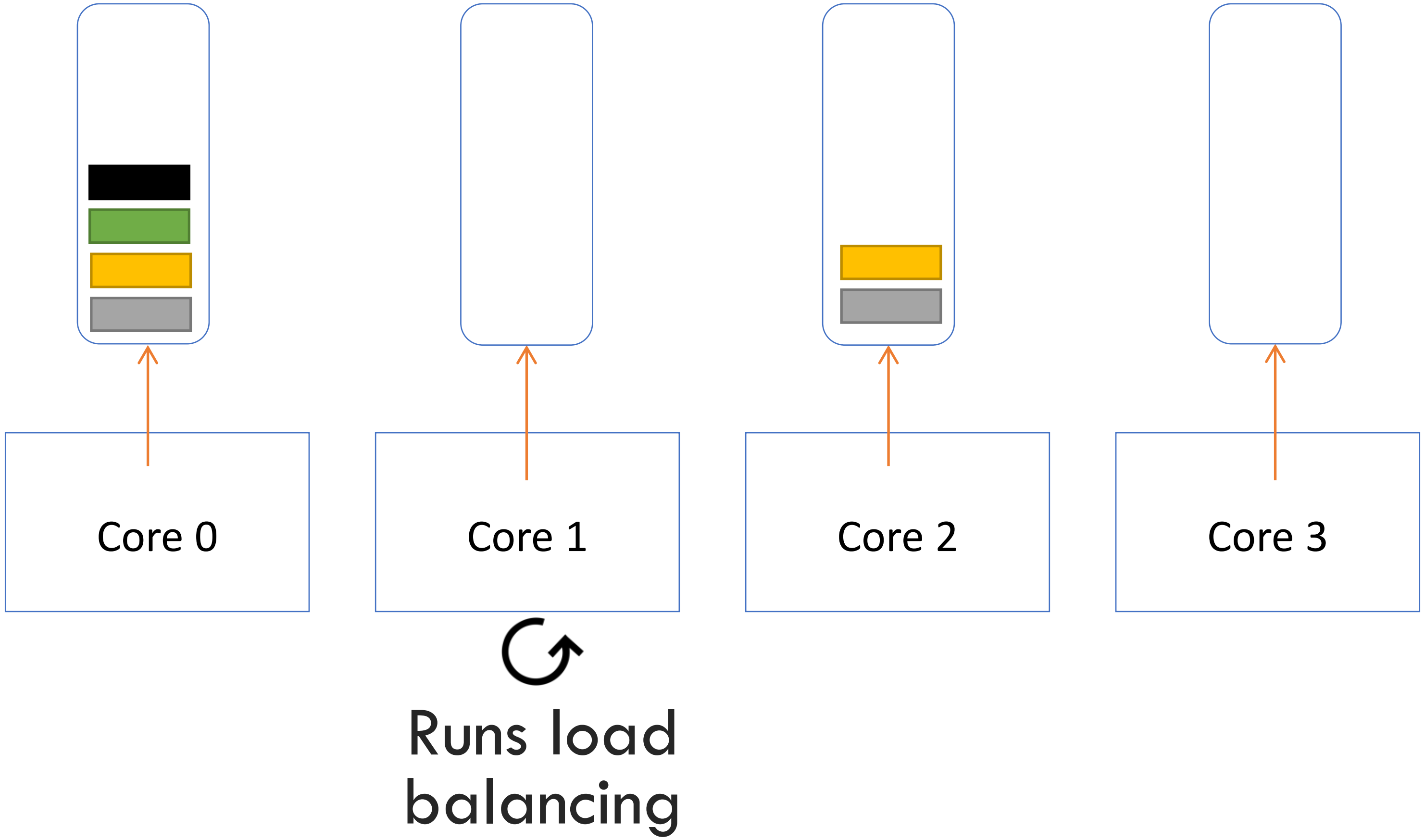
Act *(e.g., steal threads from locked core)*

Based on possibly outdated observations!

37/58

Challenge #1

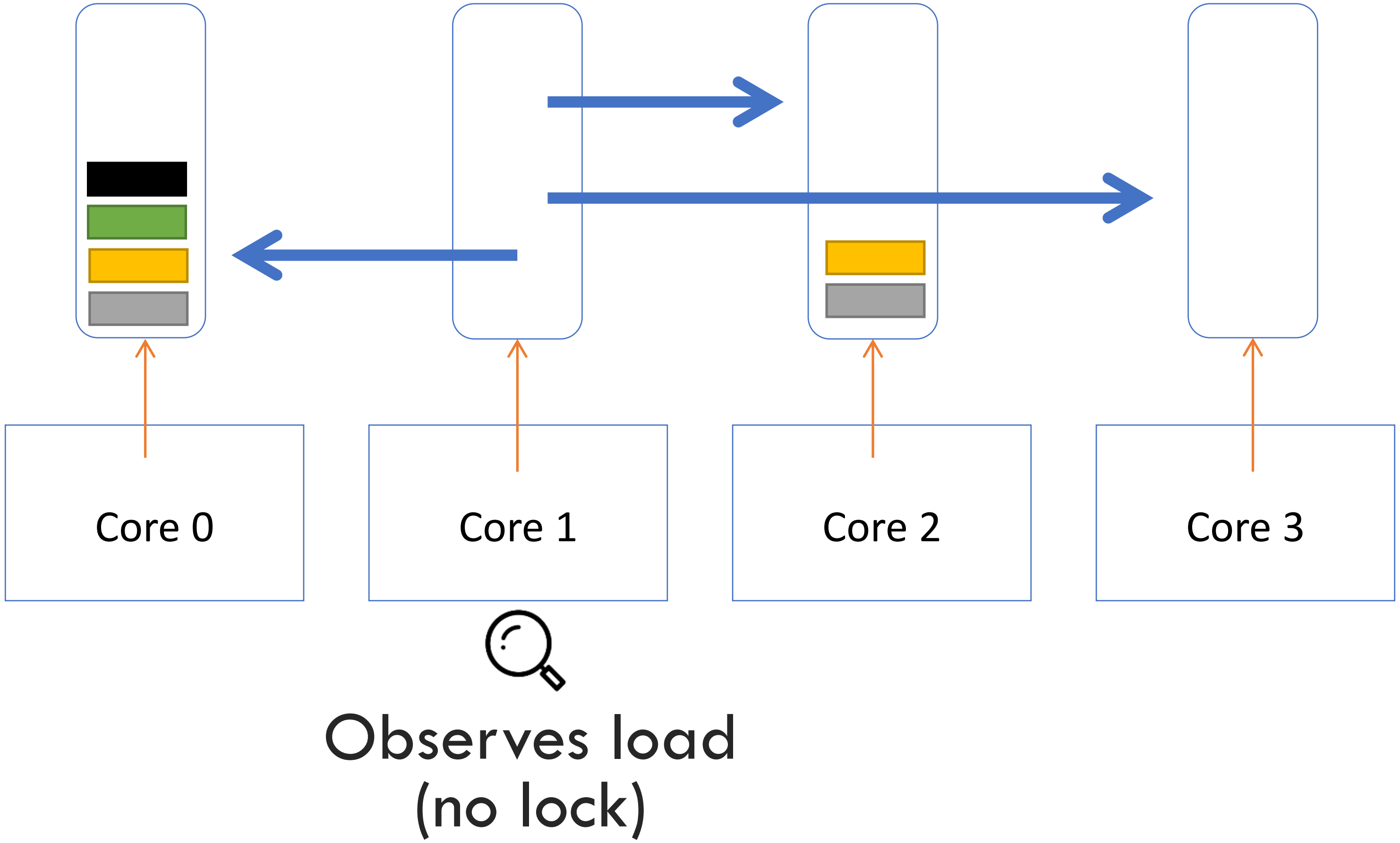
Concurrent events & optimistic concurrency



38/58

Challenge #1

Concurrent events & optimistic concurrency

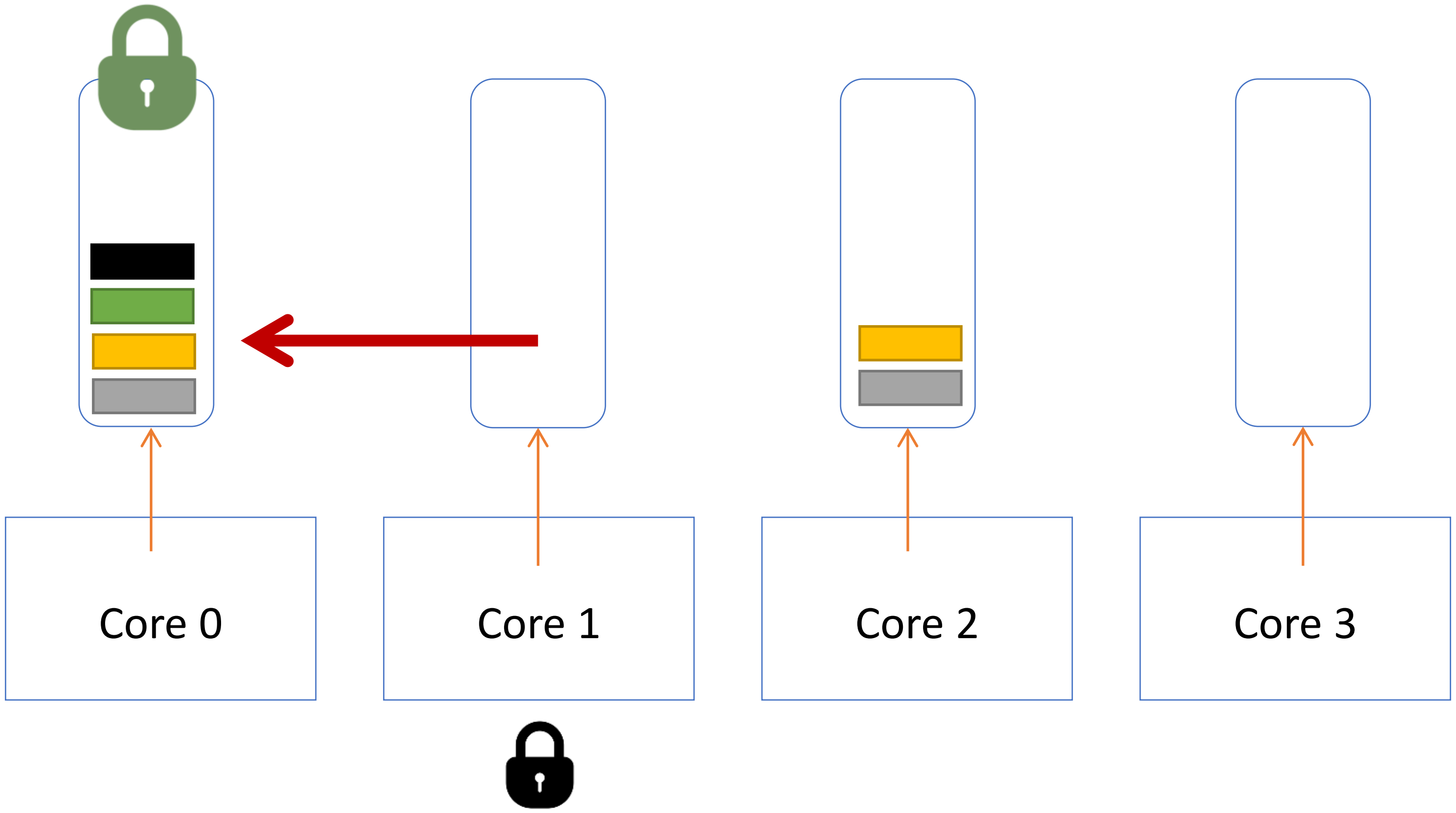


39/58

Challenge #1

Concurrent events & optimistic concurrency

Ideal scenario: no change since observations



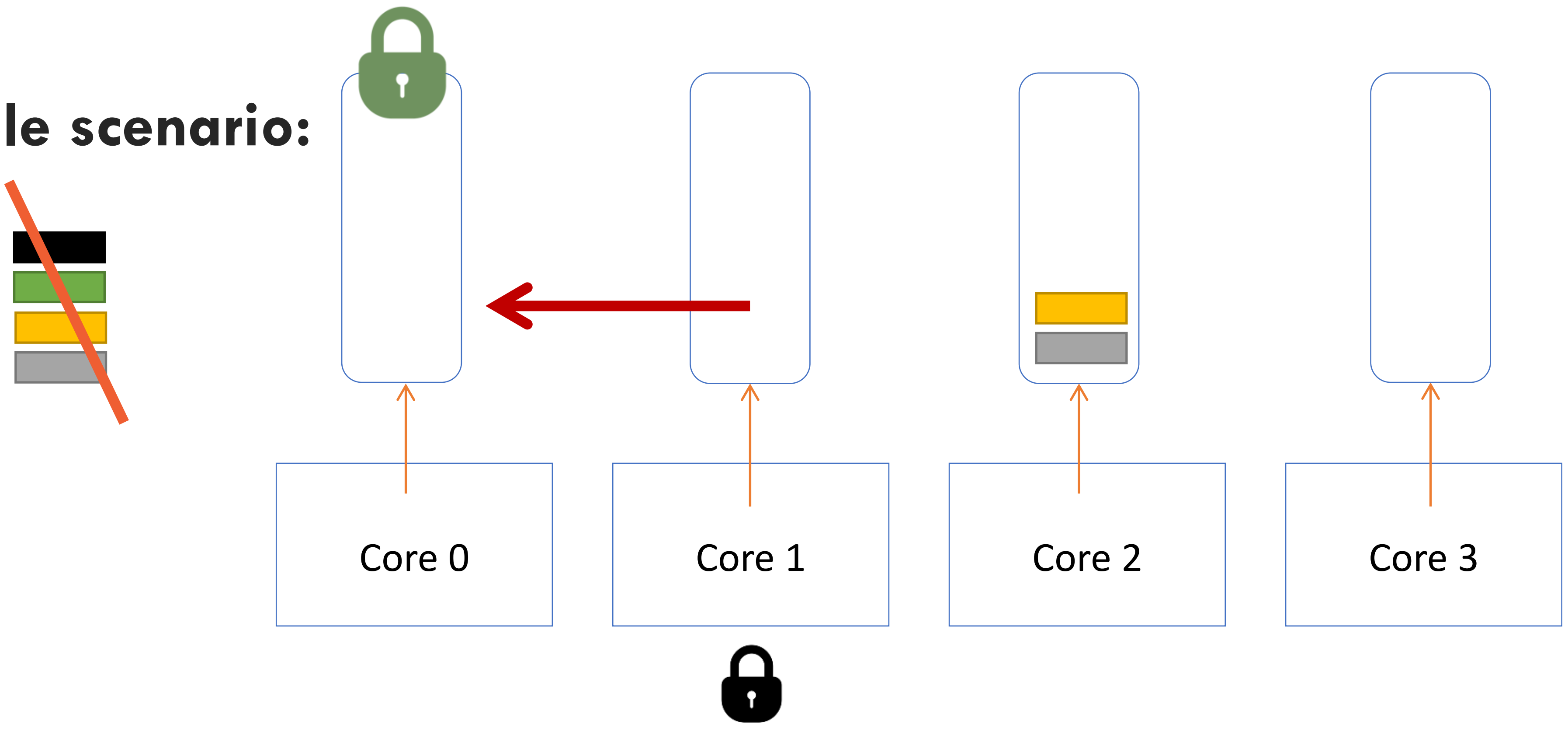
Locks **busiest**

40/58

Challenge #1

Concurrent events & optimistic concurrency

Possible scenario:



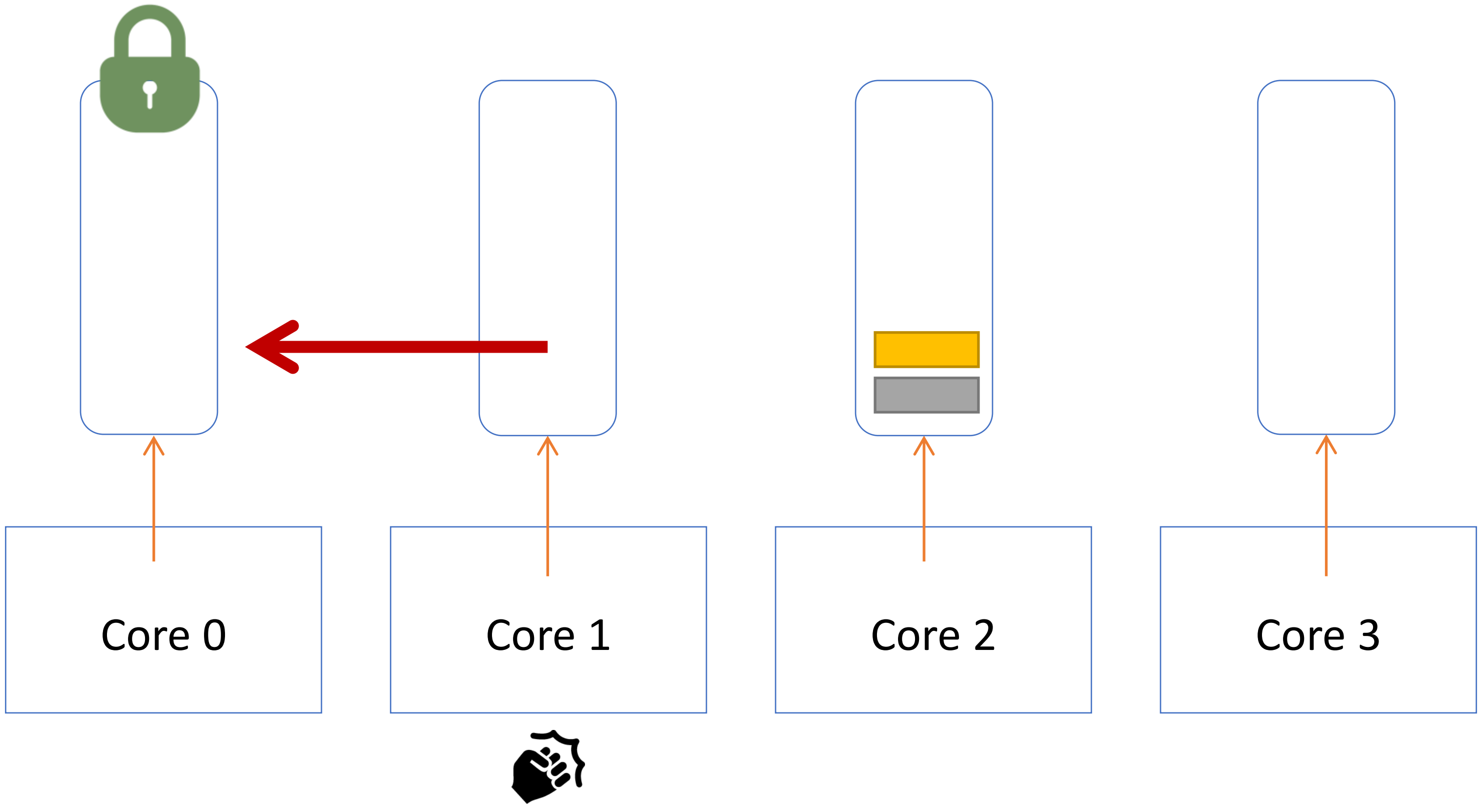
Locks **"busiest"**

Busiest might have no thread left!

41/58

Challenge #1

Concurrent events & optimistic concurrency



(Fail to)
Steal from busiest

42/58

Challenge #1

Concurrent events & optimistic concurrency

time

Observe

Lock

Act

Based on possibly outdated observations!

Definition of Work Conservation must take *concurrency* into account!

43/58

Concurrent Work Conservation Formally

Definition of overloaded with « failure cases »:

$$\exists c\ .\ (O(c) \land \neg fork(c) \land \neg unblock(c)\ \ldots)$$

If a core is overloaded

(but not because a thread was concurrently created)

44/58

Concurrent Work Conservation Formally

$$\exists c \,.\, (O(c) \wedge \neg fork(c) \wedge \neg unblock(c) \ldots) \\ \Rightarrow \\ \forall c' \,.\, \neg(I(c') \wedge \ldots)$$

45/58

Challenge #2

Existing scheduler code is hard to prove

- Schedulers handle millions of events per second

Historically: low level C code.

46/58

Challenge #2

Existing scheduler code is hard to prove

Schedulers handle millions of events per second

Historically: low level C code.

Code should be easy to prove AND efficient!

47/58

Challenge #2

Existing scheduler code is hard to prove

Schedulers handle millions of events per second

Historically: low level C code.

Code should be easy to prove AND efficient!

⇒

Domain Specific Language (DSL)

48/58

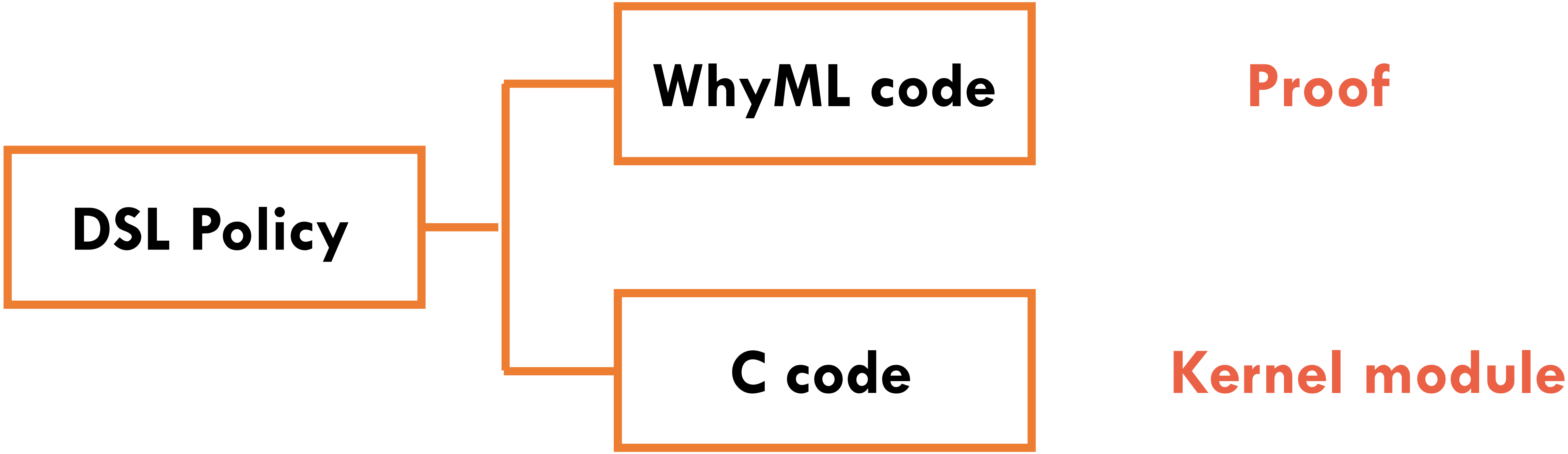
DSL advantages

Trade expressiveness for expertise/knowledge:

- Robustness: (static) verification of properties
- Explicit concurrency: explicit shared variables
- Performance: efficient compilation

49/58

DSL-based proofs



DSL: close to C
Easy learn and to compile to WhyML and C

50/58

DSL-based proofs

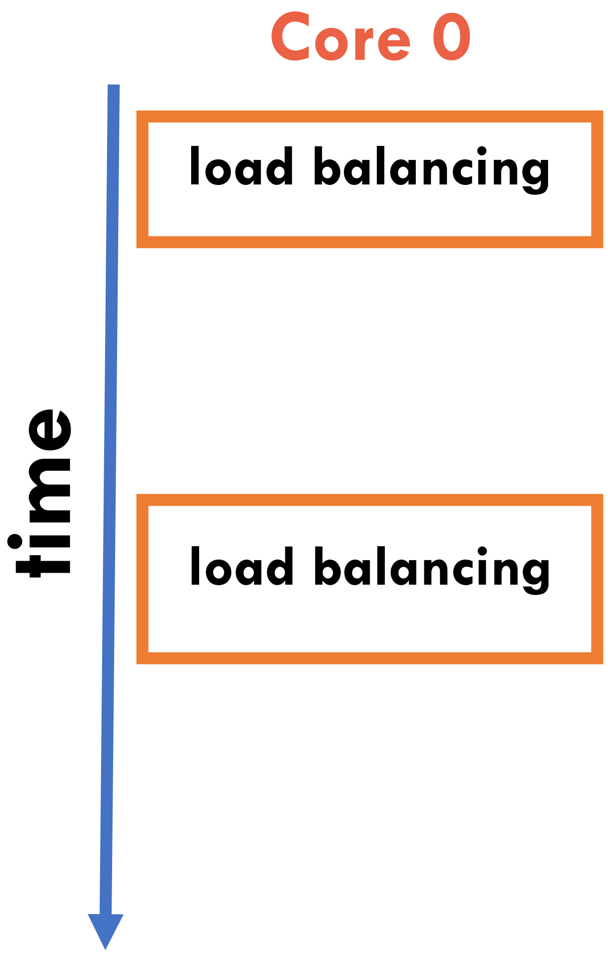
Proof on all possible interleavings

51/58

DSL-based proofs

Proof on all possible interleavings

Split code in blocks
(1 block = 1 read or write to a shared variable)



52/58

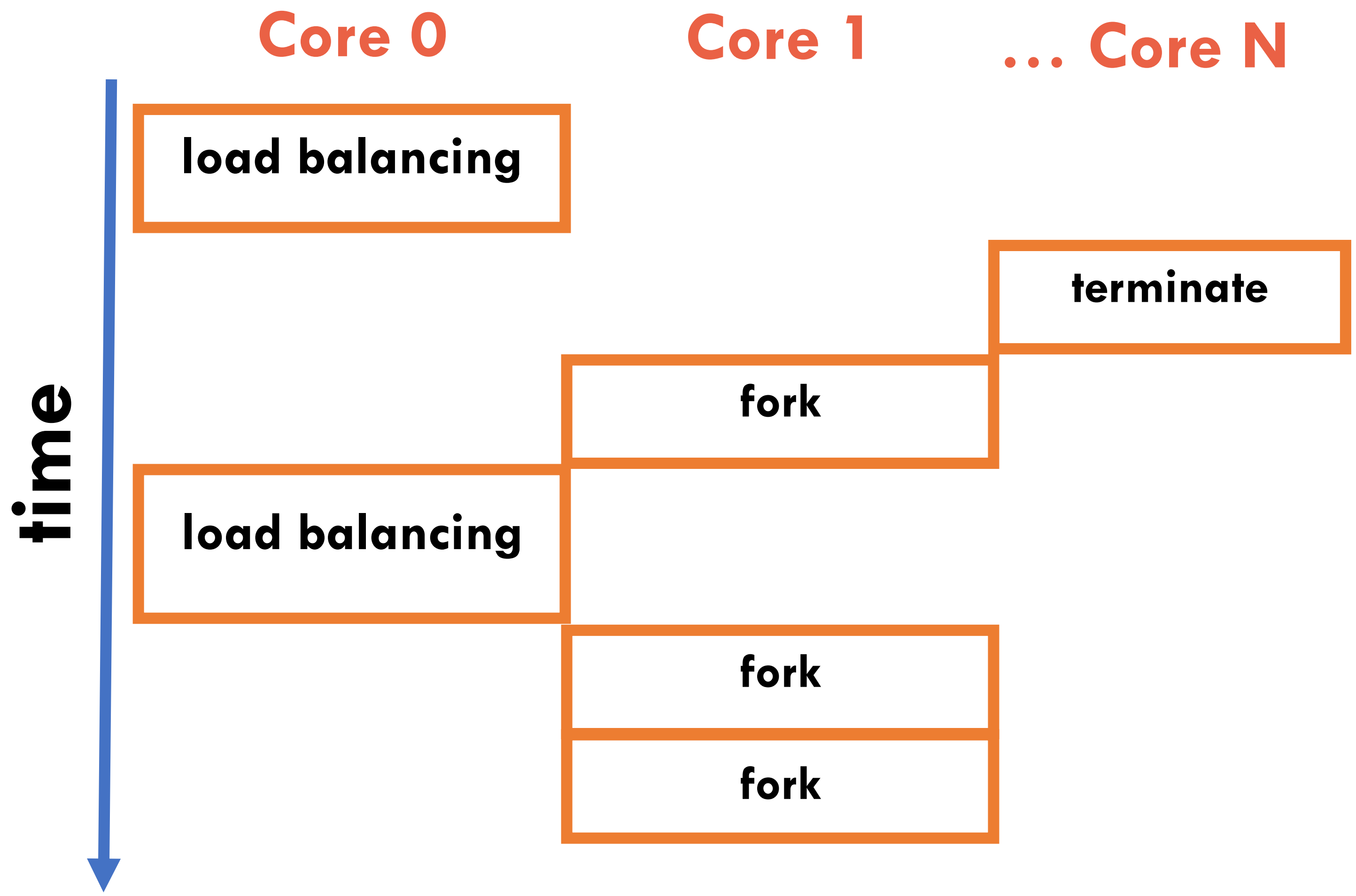
DSL-based proofs

Proof on all possible interleavings

Split code in blocks
(1 block = 1 read or write to a shared variable)

Simulate execution of concurrent blocs on N cores

Concurrent WC must hold at the end of the load balancing



53/58

DSL-based proofs

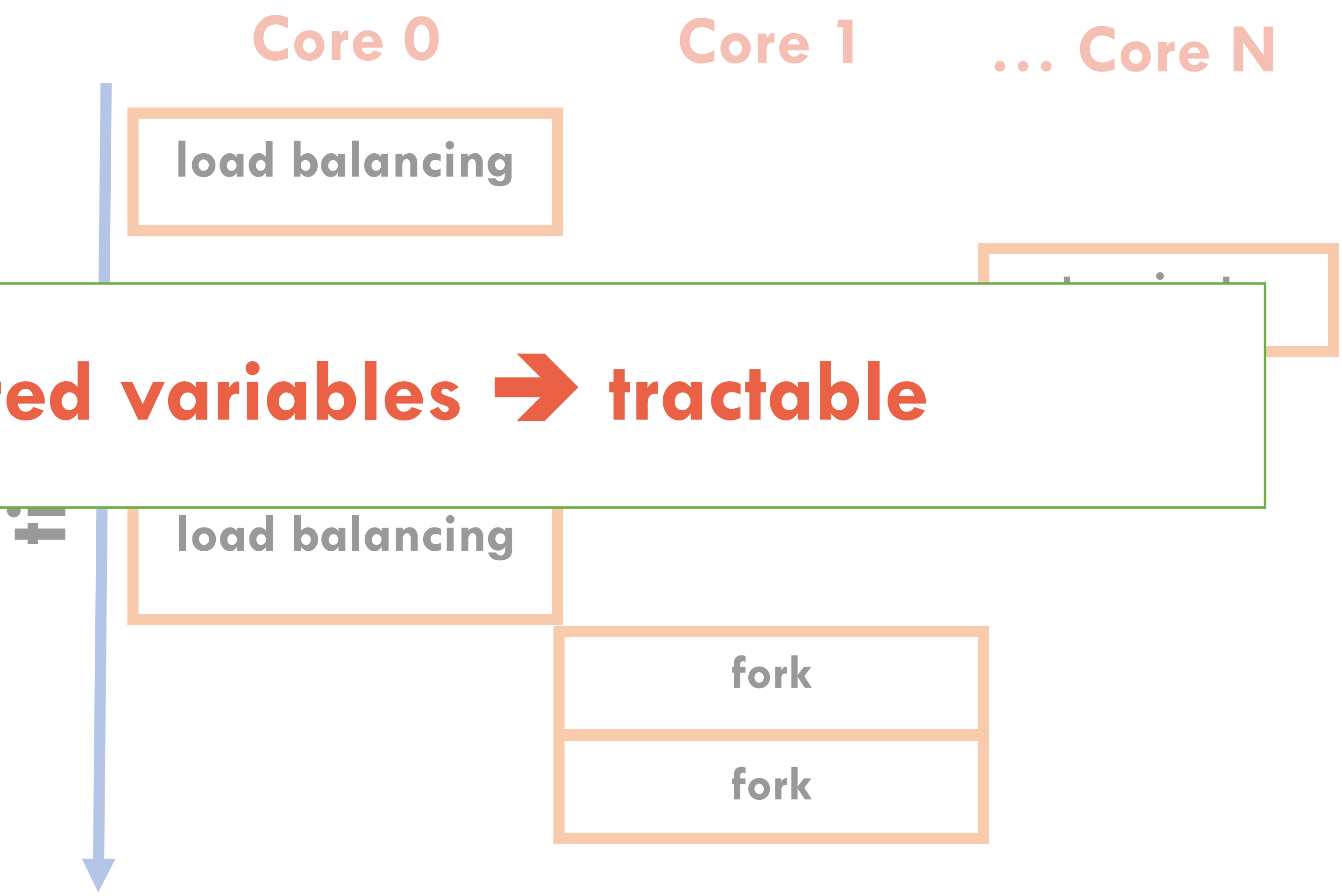
Proof on all possible interleavings

(

shared variable)

Simulate execution of concurrent blocs on N cores

Concurrent WC must always hold!



DSL → few shared variables → tractable

54/58

Evaluation

CFS-CWC (365 LOC)
Hierarchical CFS-like scheduler

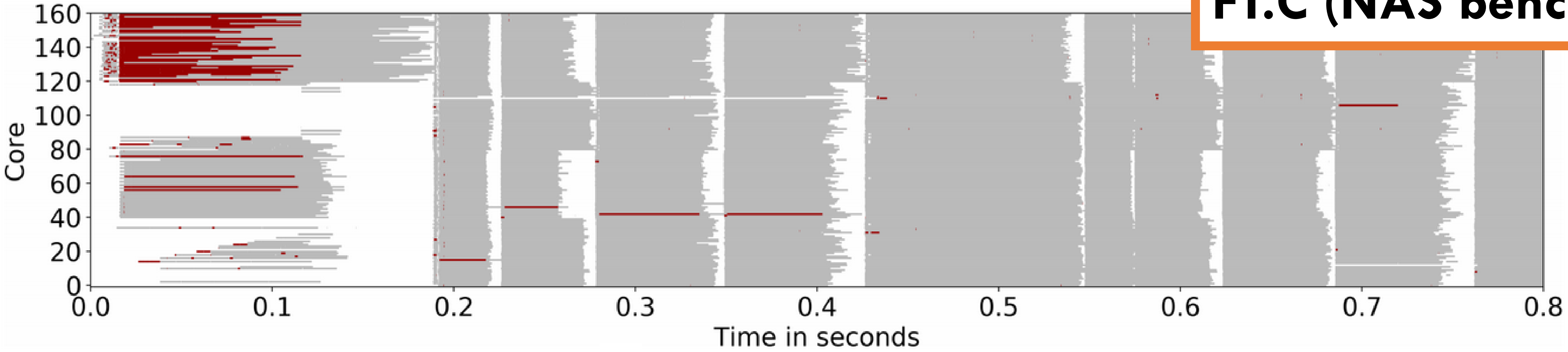
CFS-CWC-FLAT (222 LOC)
Single level CFS-like scheduler

ULE-CWC (244 LOC)
BSD-like scheduler

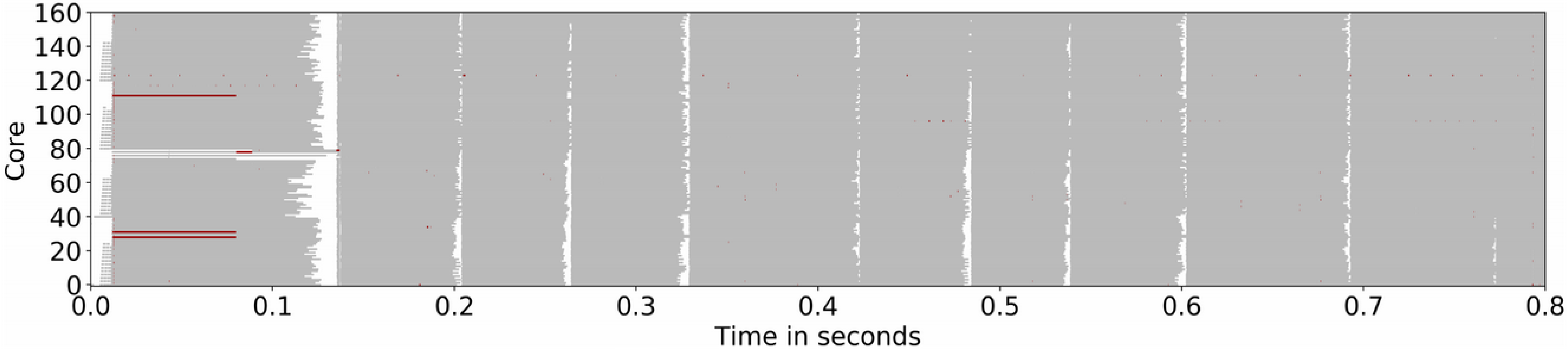
55/58

Less idle time

FT.C (NAS benchmark)

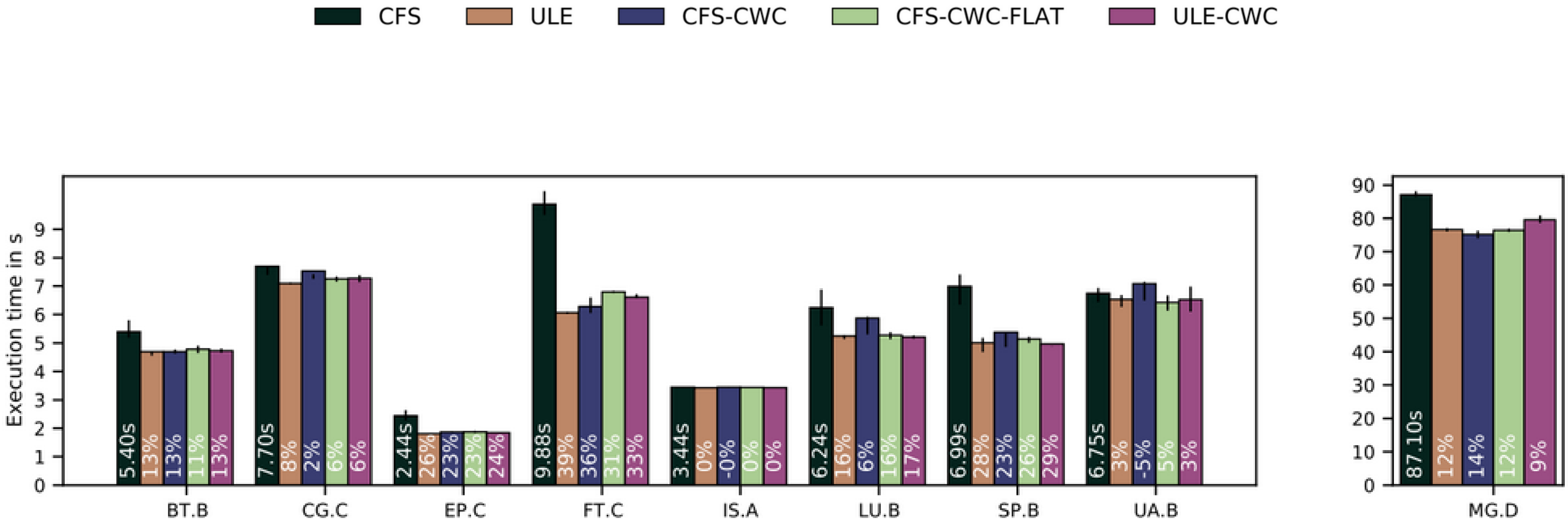


Execution with vanilla CFS.



Execution with CFS-CWC.

Comparable or better performance



NAS benchmarks (lower is better)

57/58

Conclusion

Work conservation: not straightforward!
… new formalism: *concurrent* work conservation!

Complex concurrency scheme
…proofs made tractable using a DSL.

Performance: similar or better than CFS.

58/58