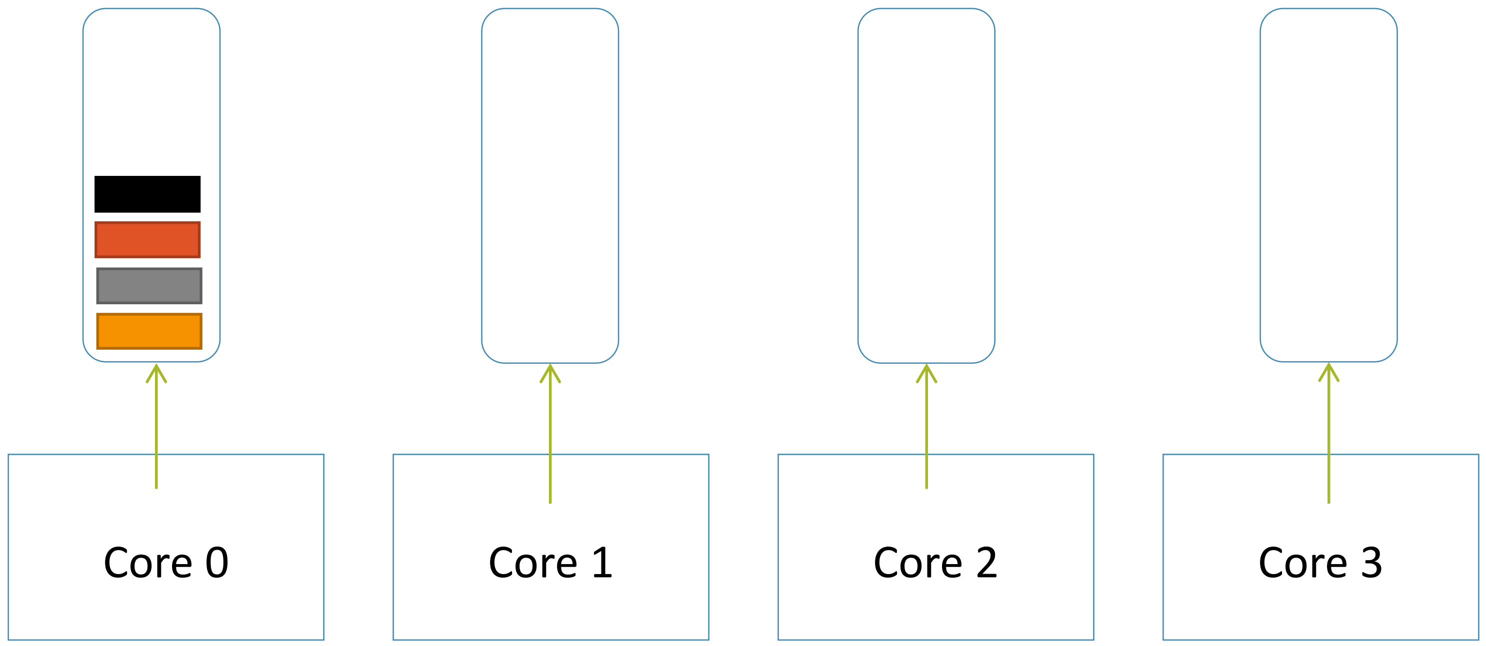
Provable Multicore Schedulers with Ipanema: Application to Work-Conservation

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

Work conservation

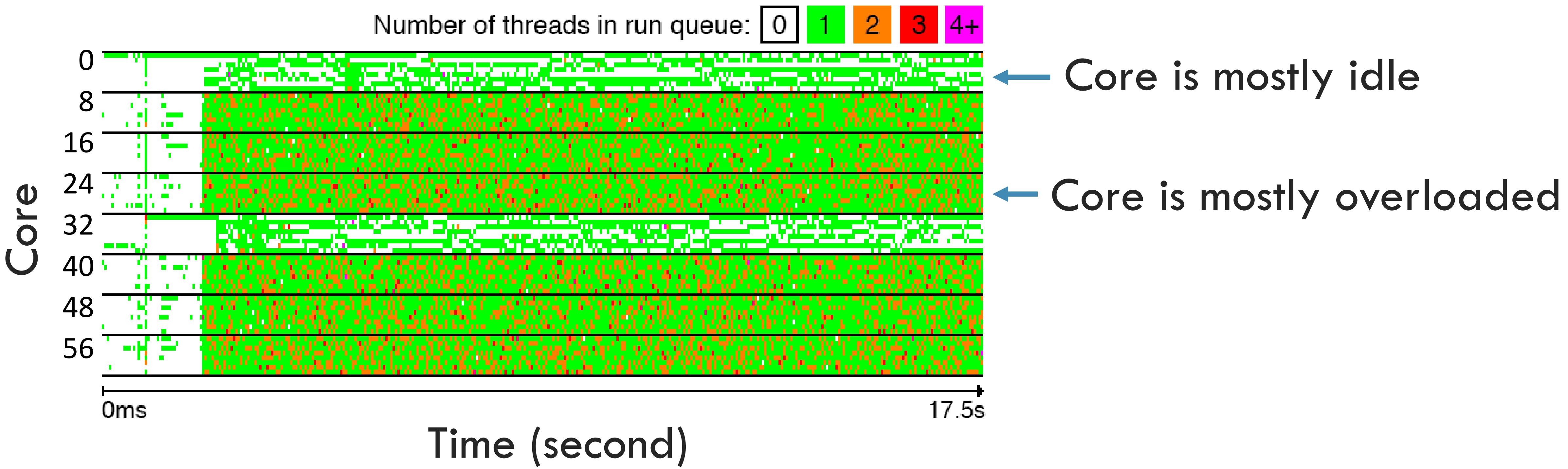
"No core should be left idle when a core is overloaded"



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

Problem

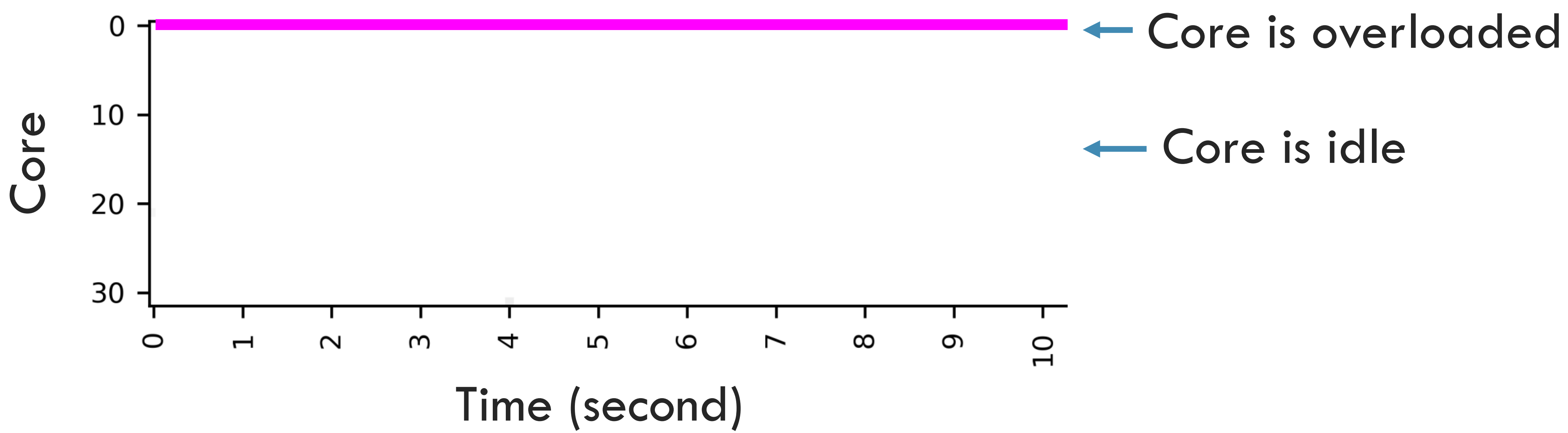
Linux (CFS) suffers from work conservation issues



[Lozi et al. 2016]

Problem

FreeBSD (ULE) suffers from work conservation issues



[Bouron et al. 2018]

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]

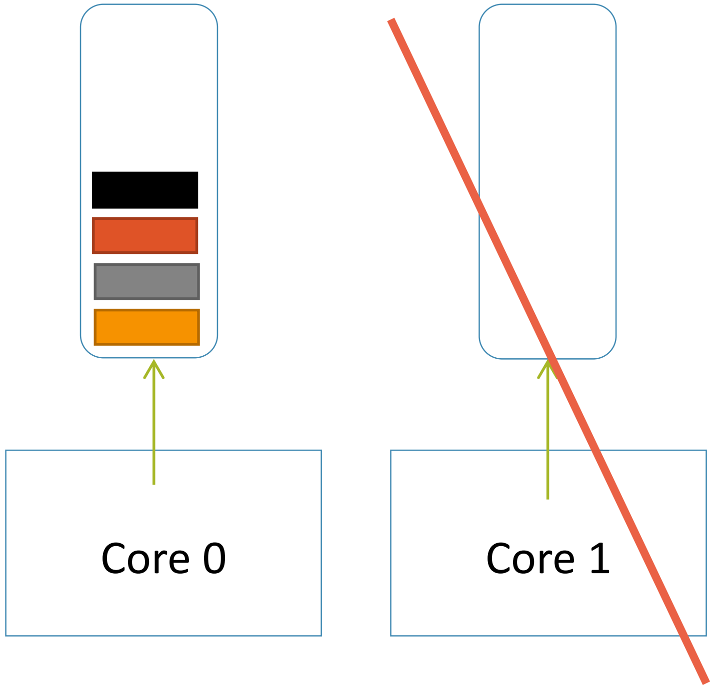
This talk

Formally prove work-conservation

Work Conservation Formally

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

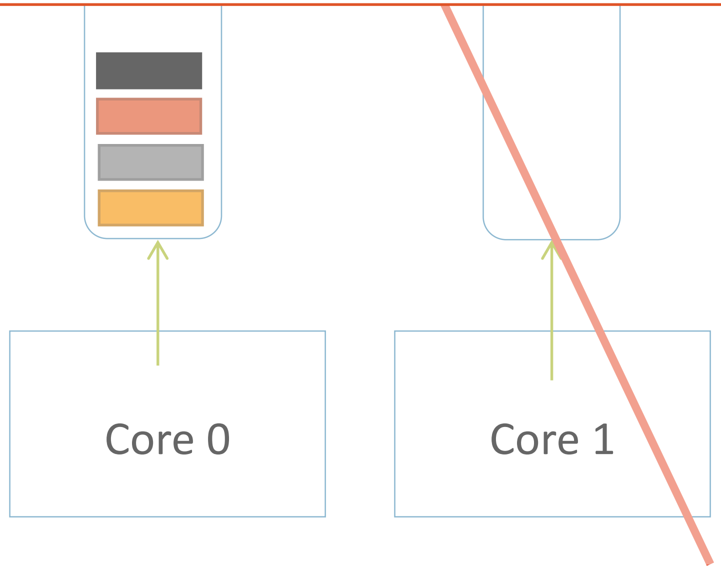
If a core is overloaded, no core is idle



Work Conservation Formally

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

Does not work for realistic schedulers!



Challenge #1

Concurrent events & optimistic concurrency

Challenge #1

Concurrent events & optimistic concurrency

time

Observe (state of **every** core)

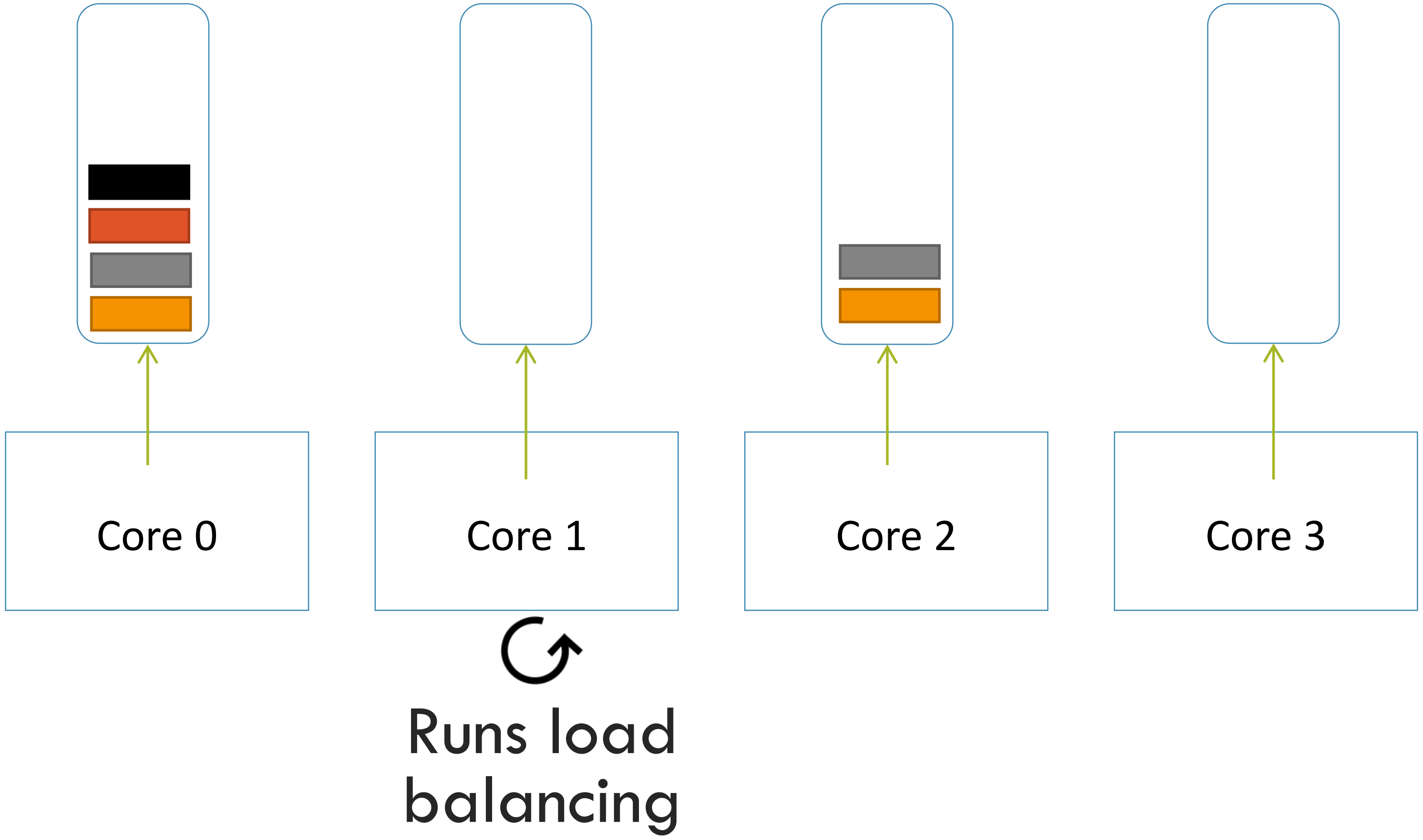
Lock (**one** core – less overhead)

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

Based on possibly outdated observations!

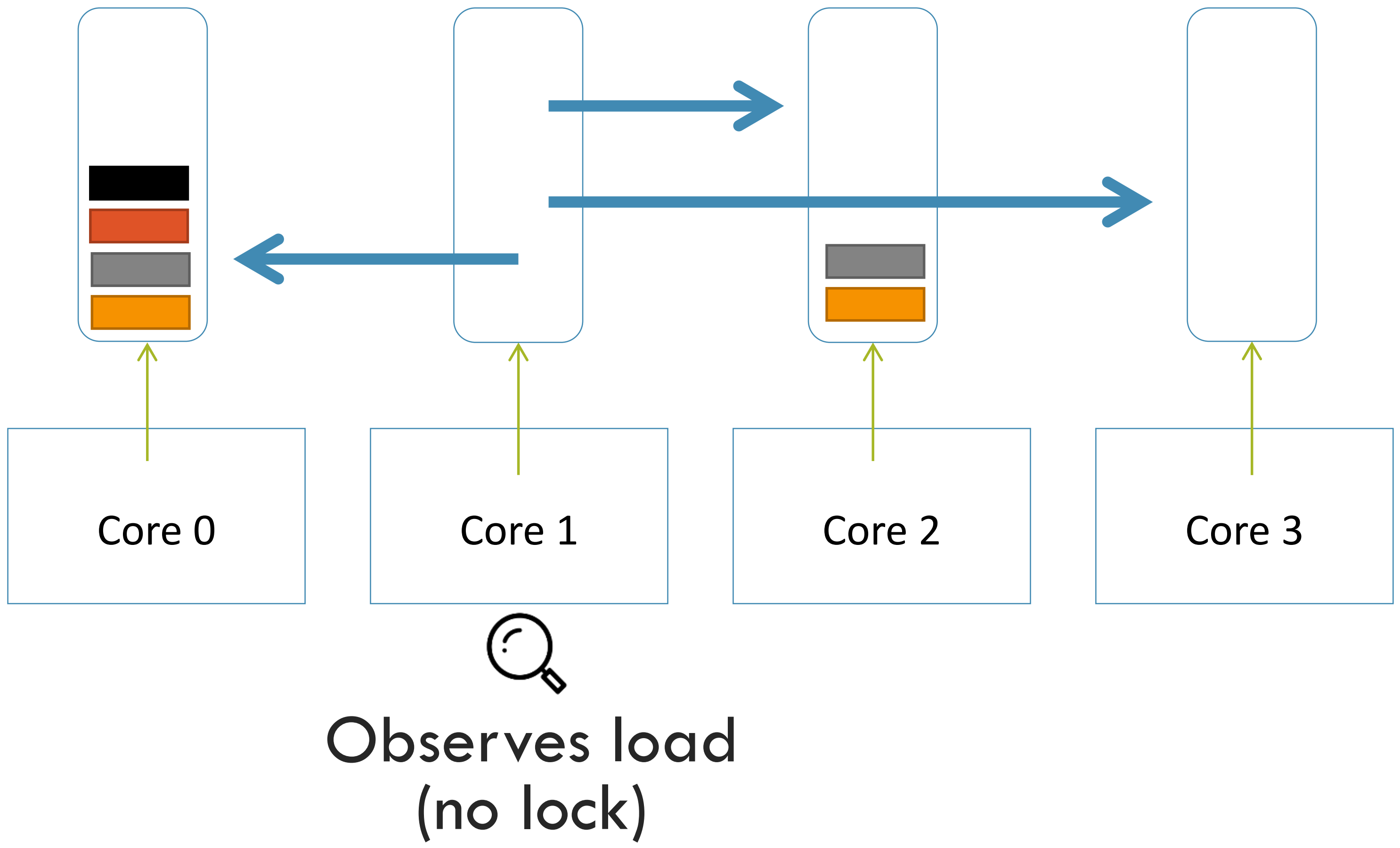
Challenge #1

Concurrent events & optimistic concurrency



Challenge #1

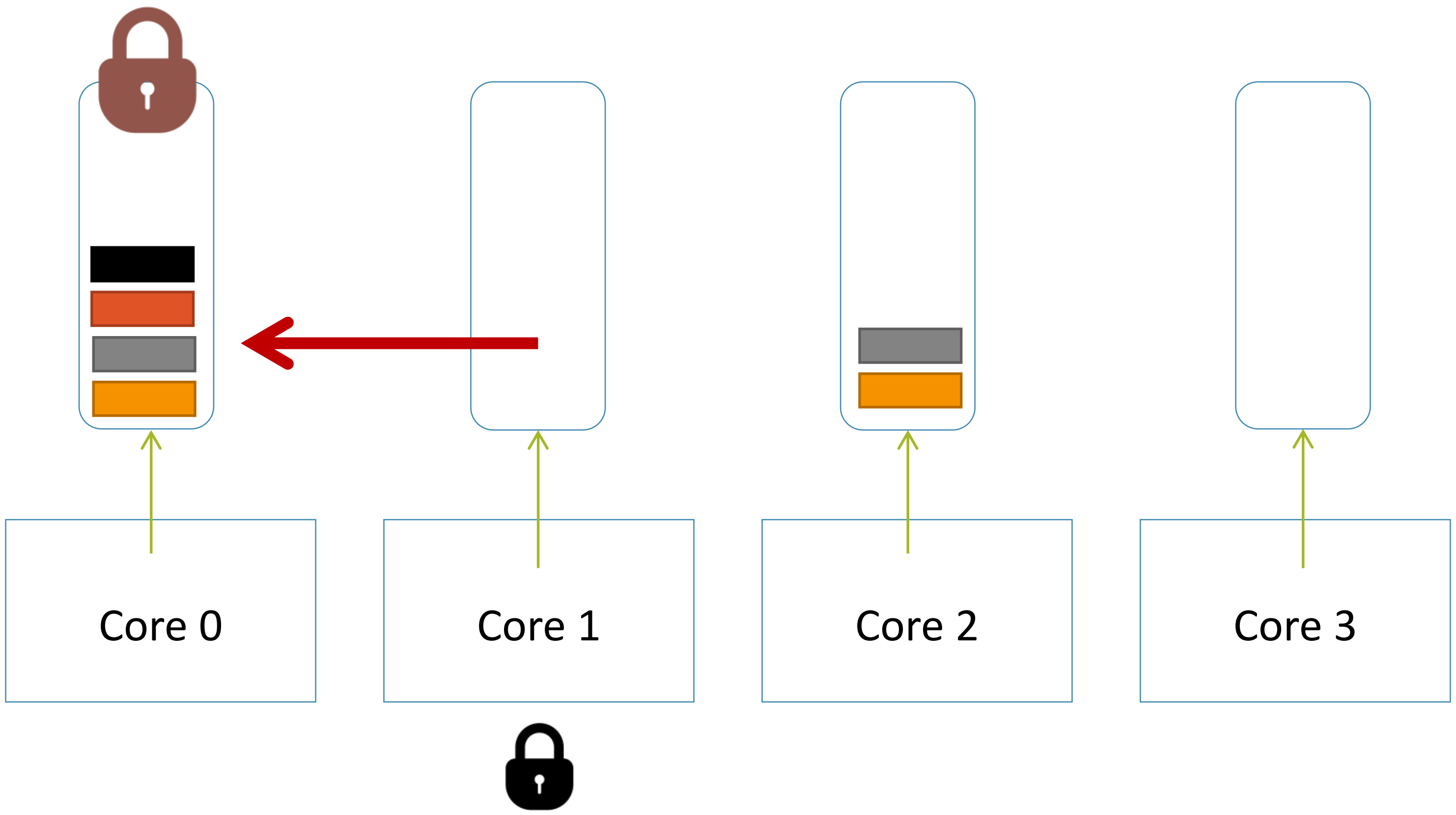
Concurrent events & optimistic concurrency



Challenge #1

Concurrent events & optimistic concurrency

Ideal scenario: no change since observations

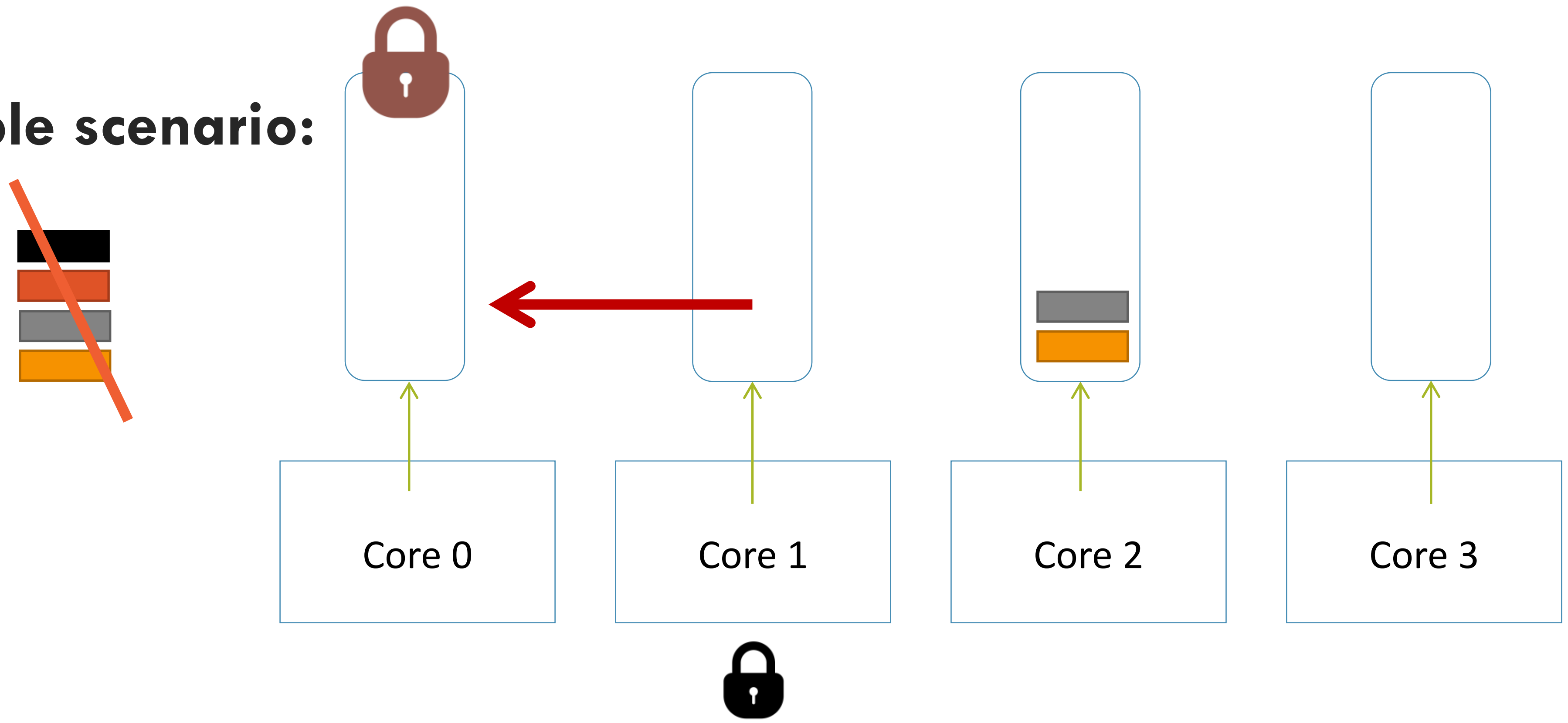


Locks **busiest**

Challenge #1

Concurrent events & optimistic concurrency

Possible scenario:

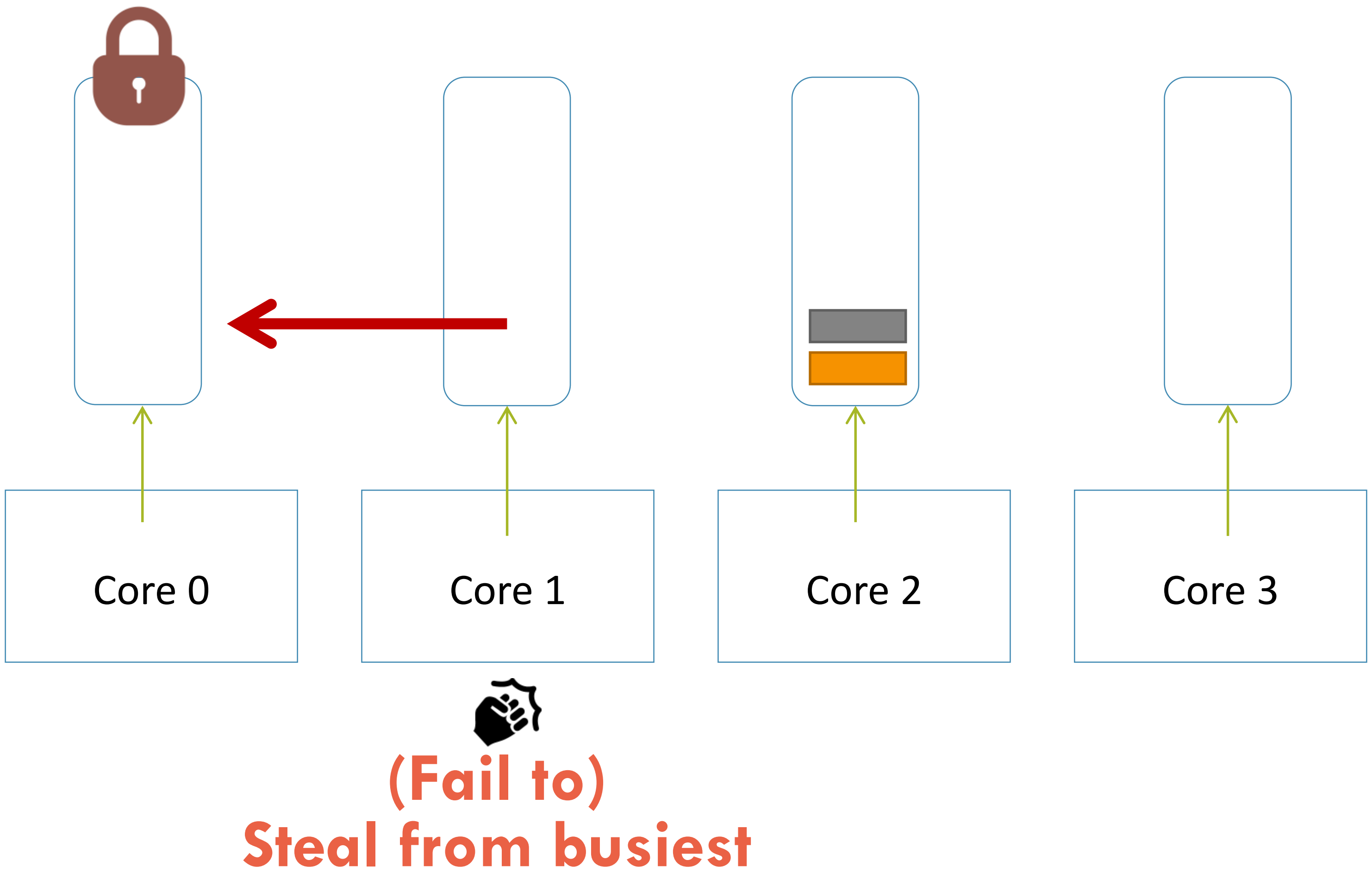


Locks **"busiest"**

Busiest might have no thread left! (Concurrent blocks/terminations.)

Challenge #1

Concurrent events & optimistic concurrency



Challenge #1

Concurrent events & optimistic concurrency

time

Observe

Lock

Act

Based on possibly outdated observations!

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

Concurrent Work Conservation Formally

Definition of overloaded with « failure cases »:

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

If a core is overloaded

(but not because a thread was concurrently created)

Concurrent Work Conservation Formally

$$\exists c\ .\ (O(c) \land \neg \mathrm{fork}(c) \land \neg \mathrm{unblock}(c)\ \ldots)$$
$$\Rightarrow$$
$$\forall c'\ .\ \neg(I(c') \land \ldots)$$

Challenge #2

Existing scheduler code is hard to prove

Schedulers handle millions of events per second

Historically: low level C code.

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!

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!

$\Rightarrow$

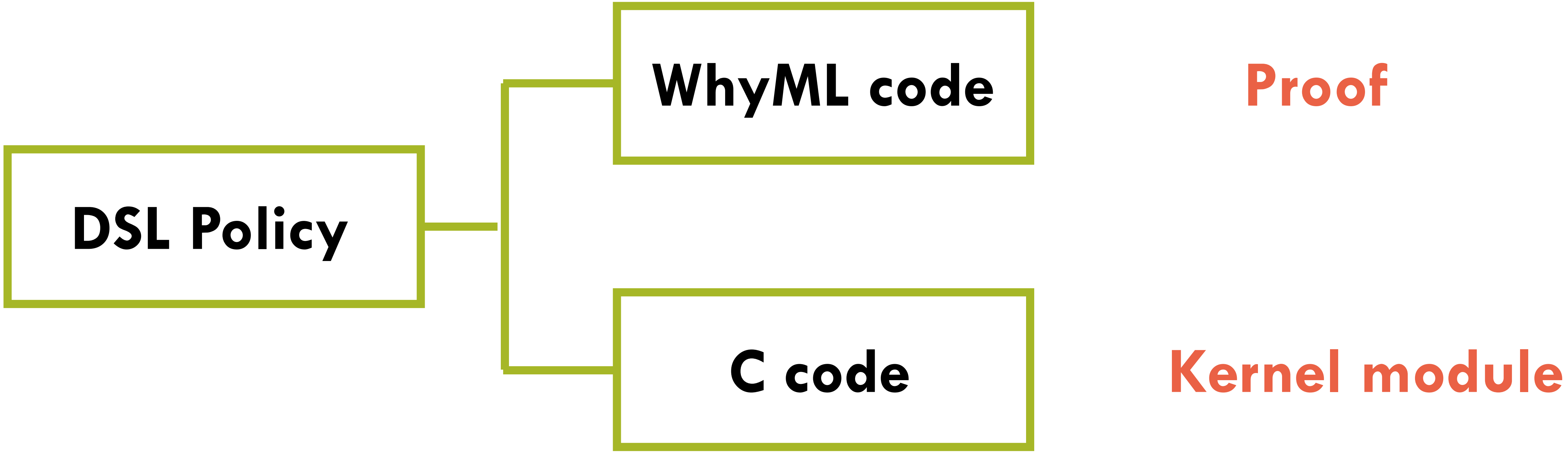
Domain Specific Language (DSL)

DSL advantages

Trade expressiveness for expertise/knowledge:

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

DSL-based proofs



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

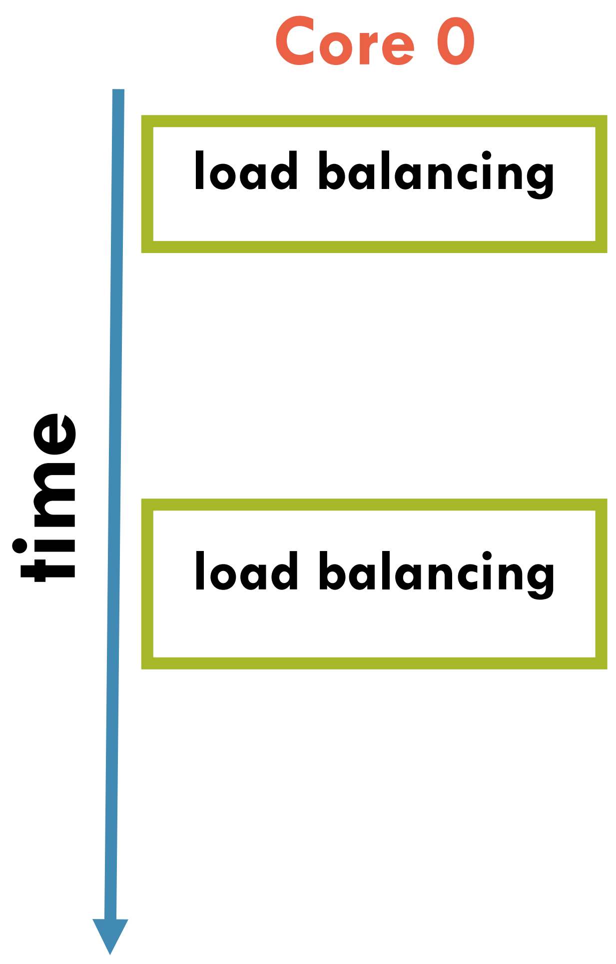
DSL-based proofs

Proof on all possible interleavings

DSL-based proofs

Proof on all possible interleavings

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



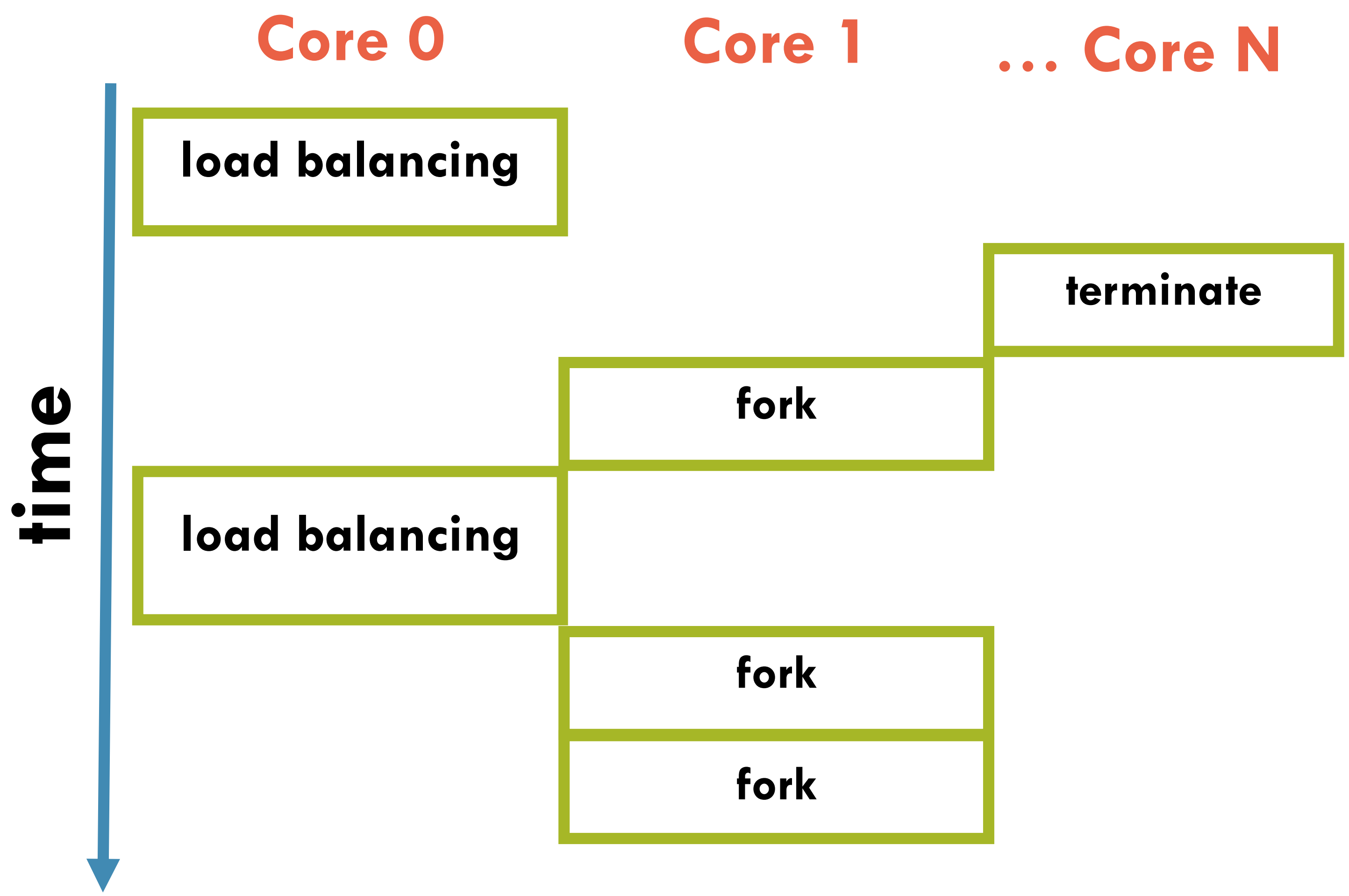
DSL-based proofs

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



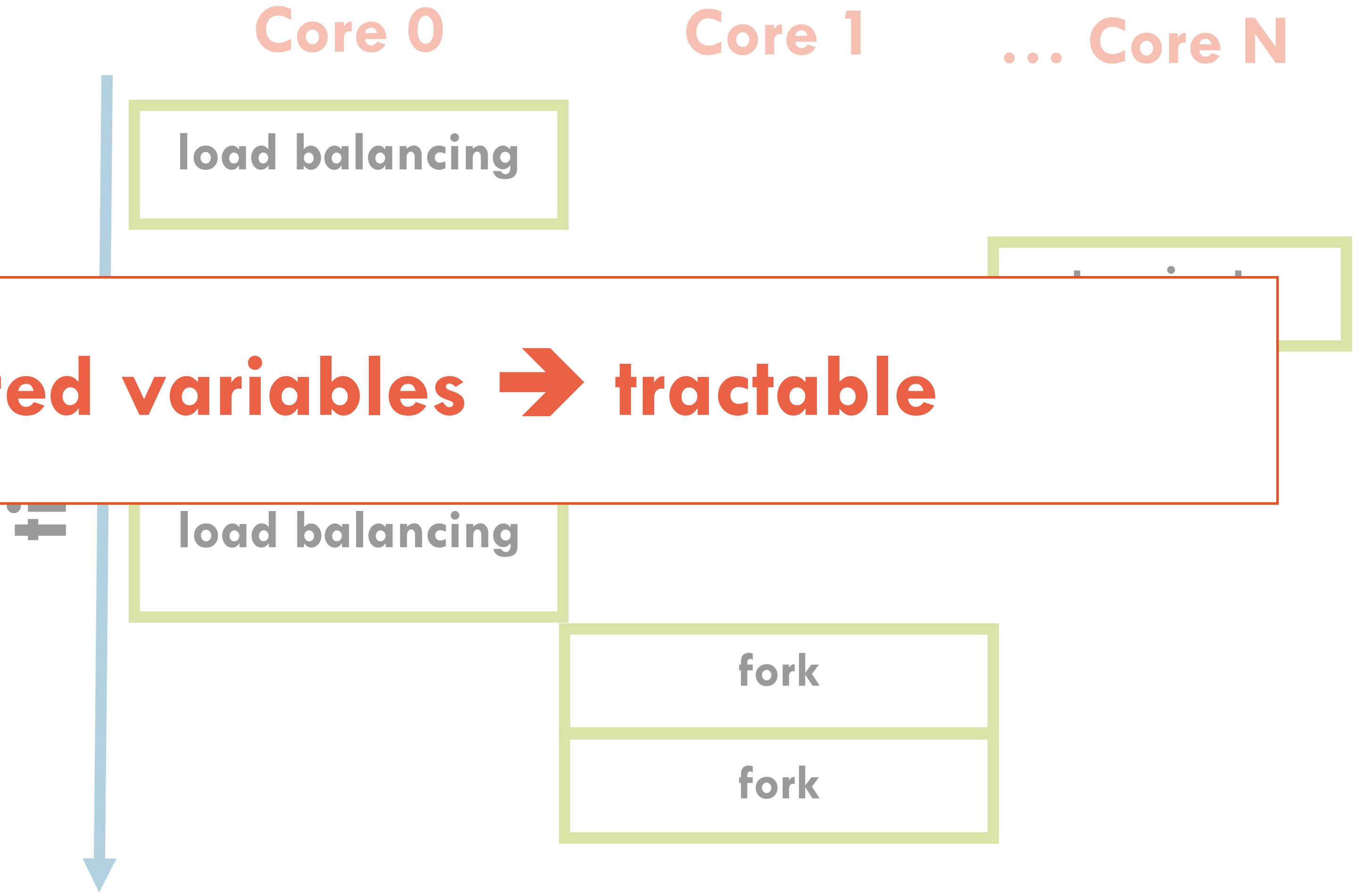
DSL-based proofs

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

Evaluation

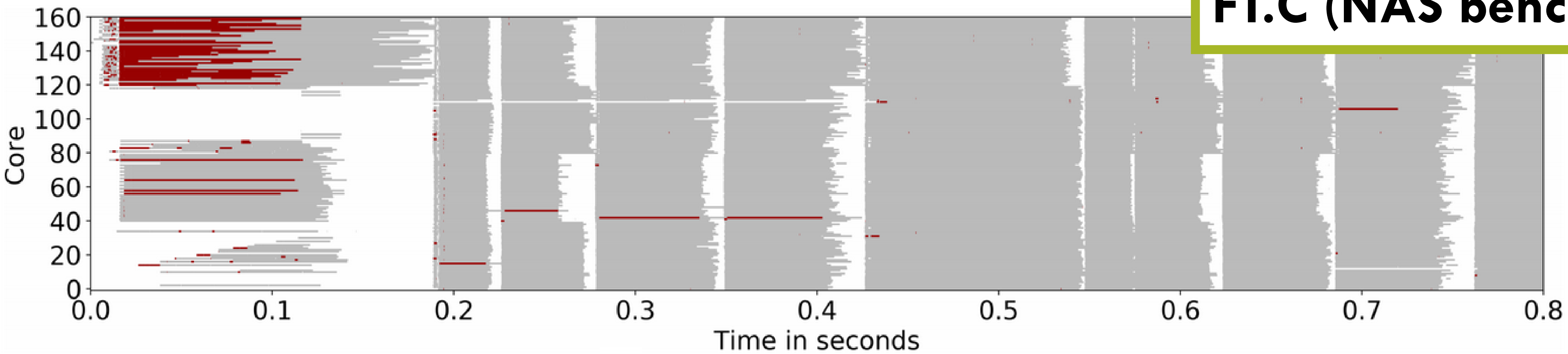
CFS-CWC (365 LOC)
Hierarchical CFS-like scheduler

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

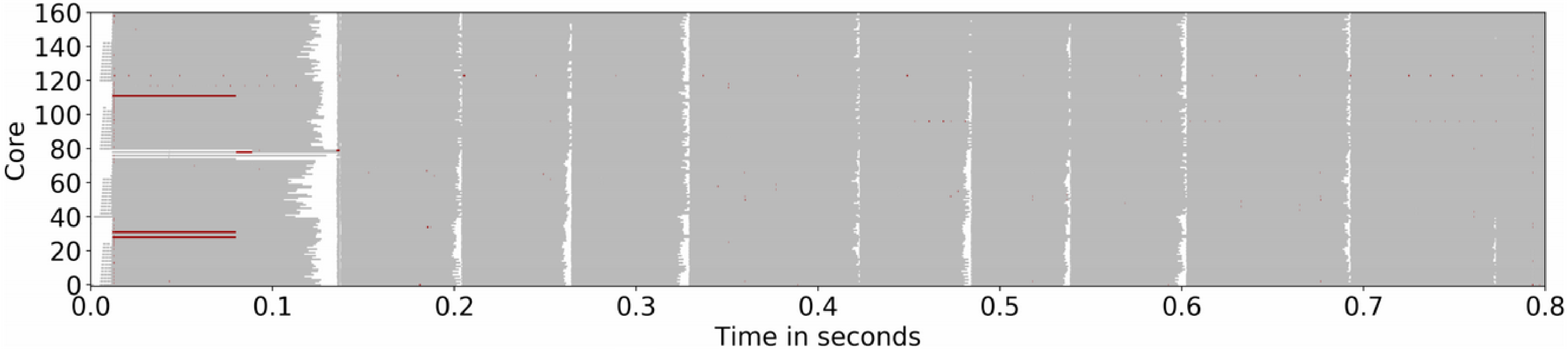
ULE-CWC (244 LOC)
BSD-like scheduler

Less idle time

FT.C (NAS benchmark)

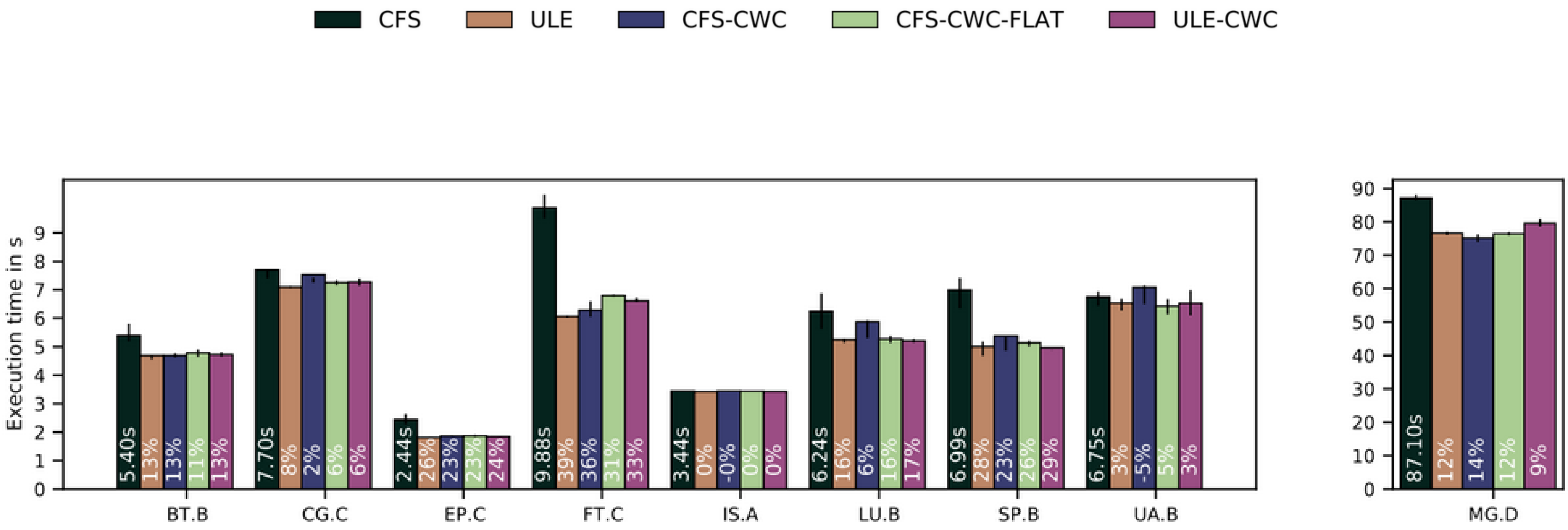


Execution with vanilla CFS.



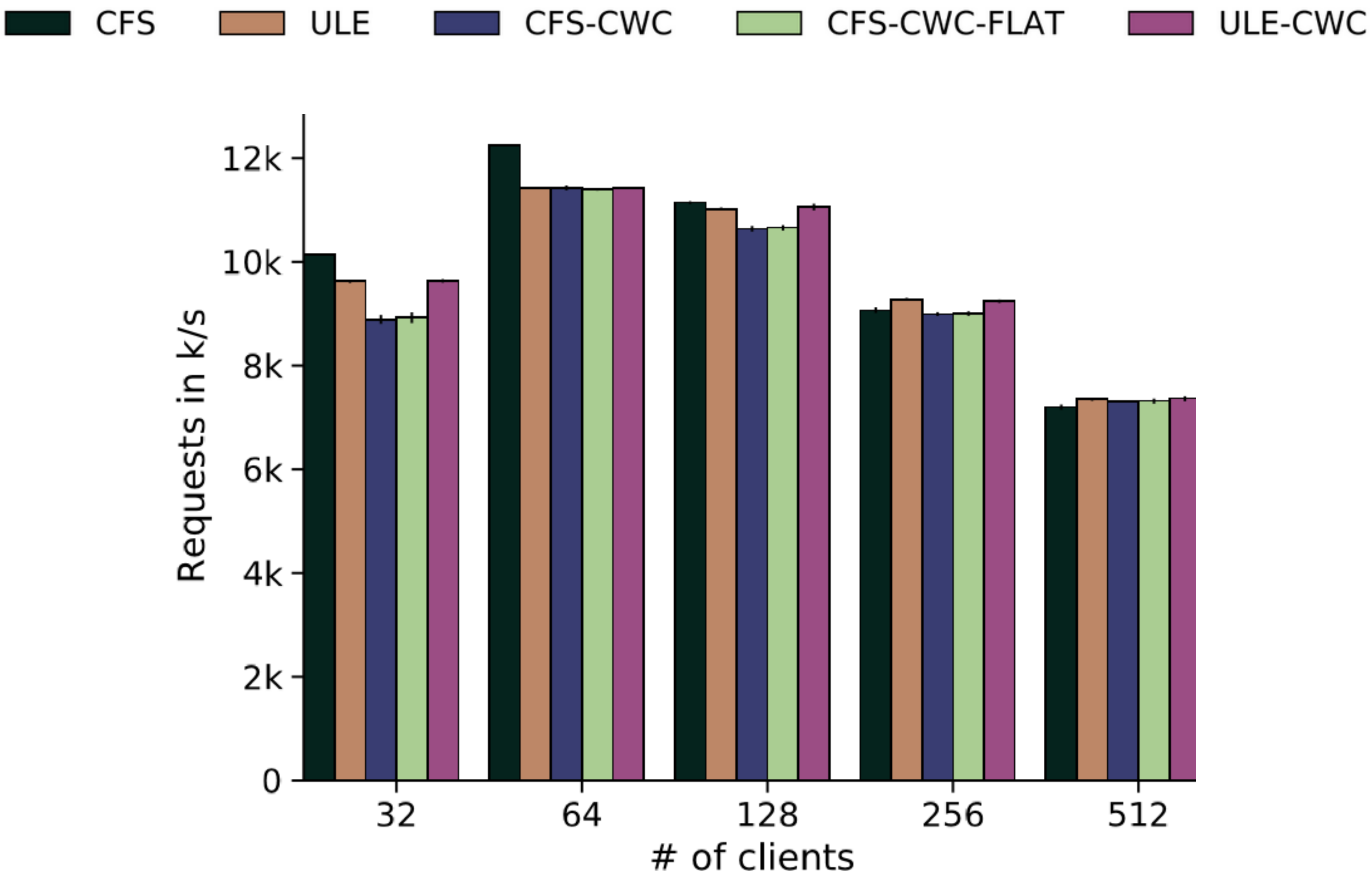
Execution with CFS-CWC.

Comparable or better performance



NAS benchmarks (lower is better)

Comparable or better performance



Sysbench on MySQL (higher is better)

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.