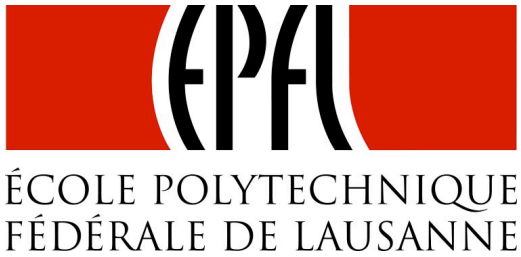
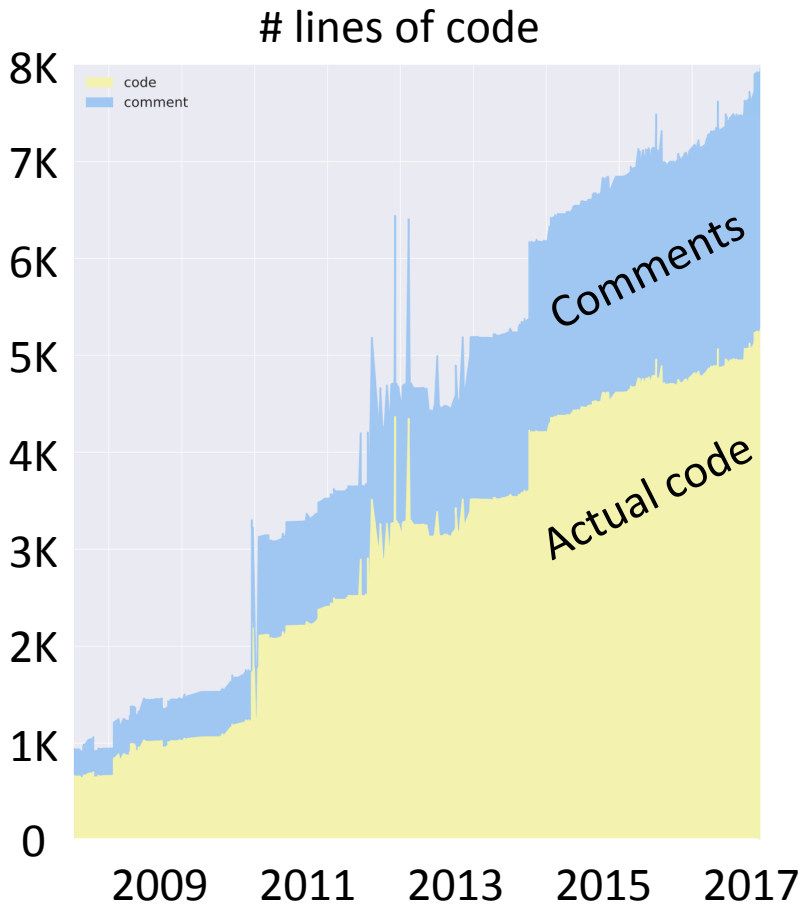
Formal verification of multicore schedulers

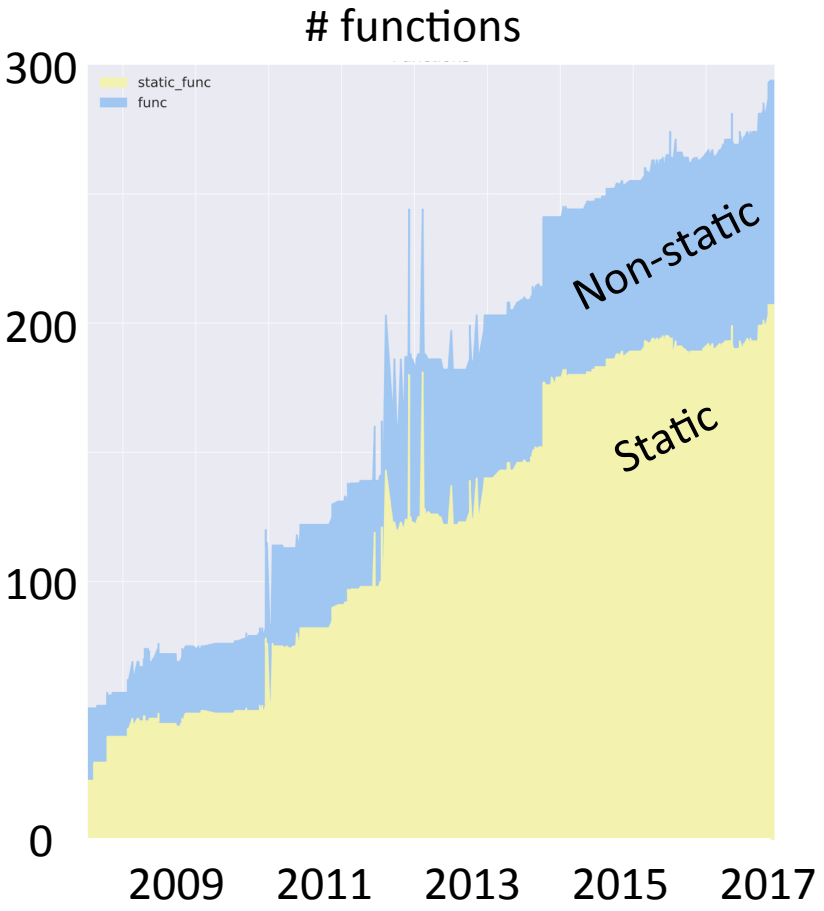
Baptiste Lepers, Jean-Pierre Lozi, Nicolas Palix, Redha Gouicem, Julien Sopena, Gilles Muller, Julia Lawall, Willy Zwaenepoel



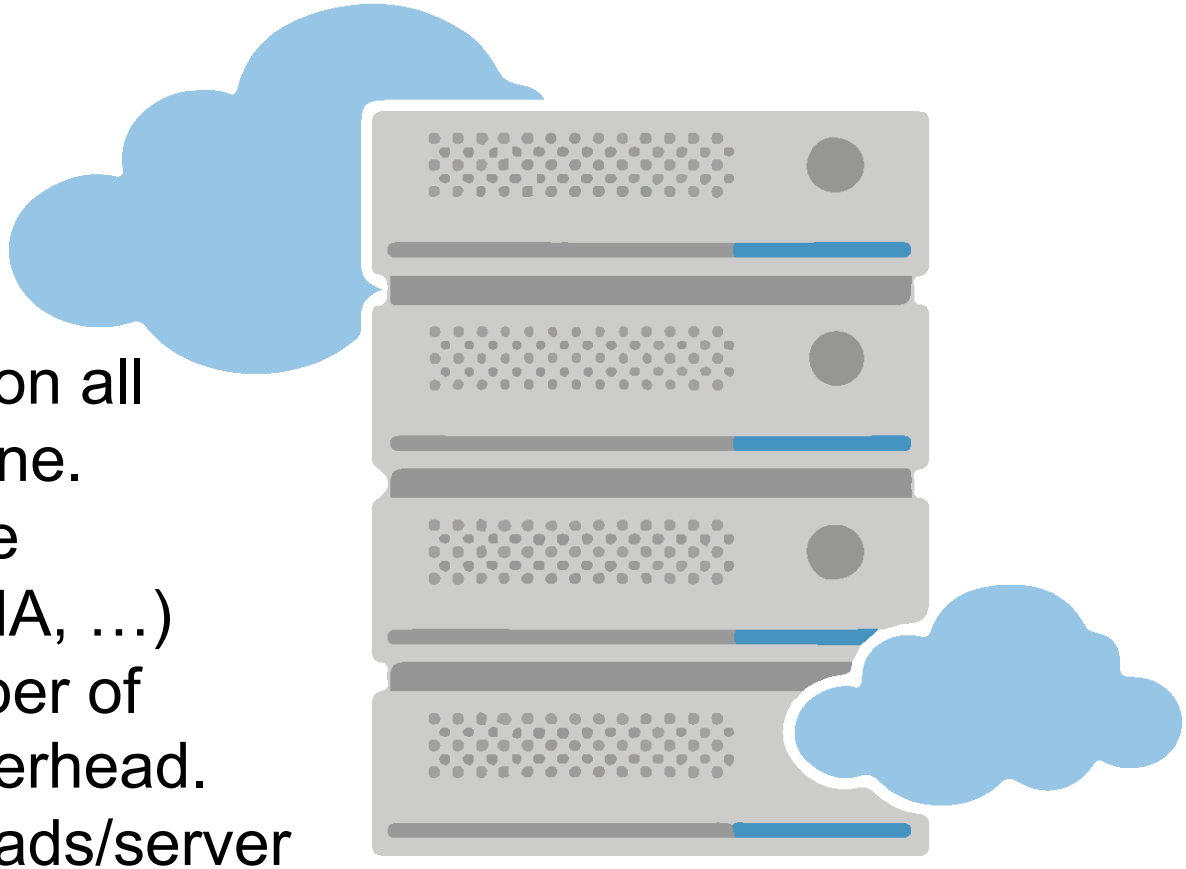
Growing complexity of scheduling

For instance, in Linux:





Roles of multicore schedulers



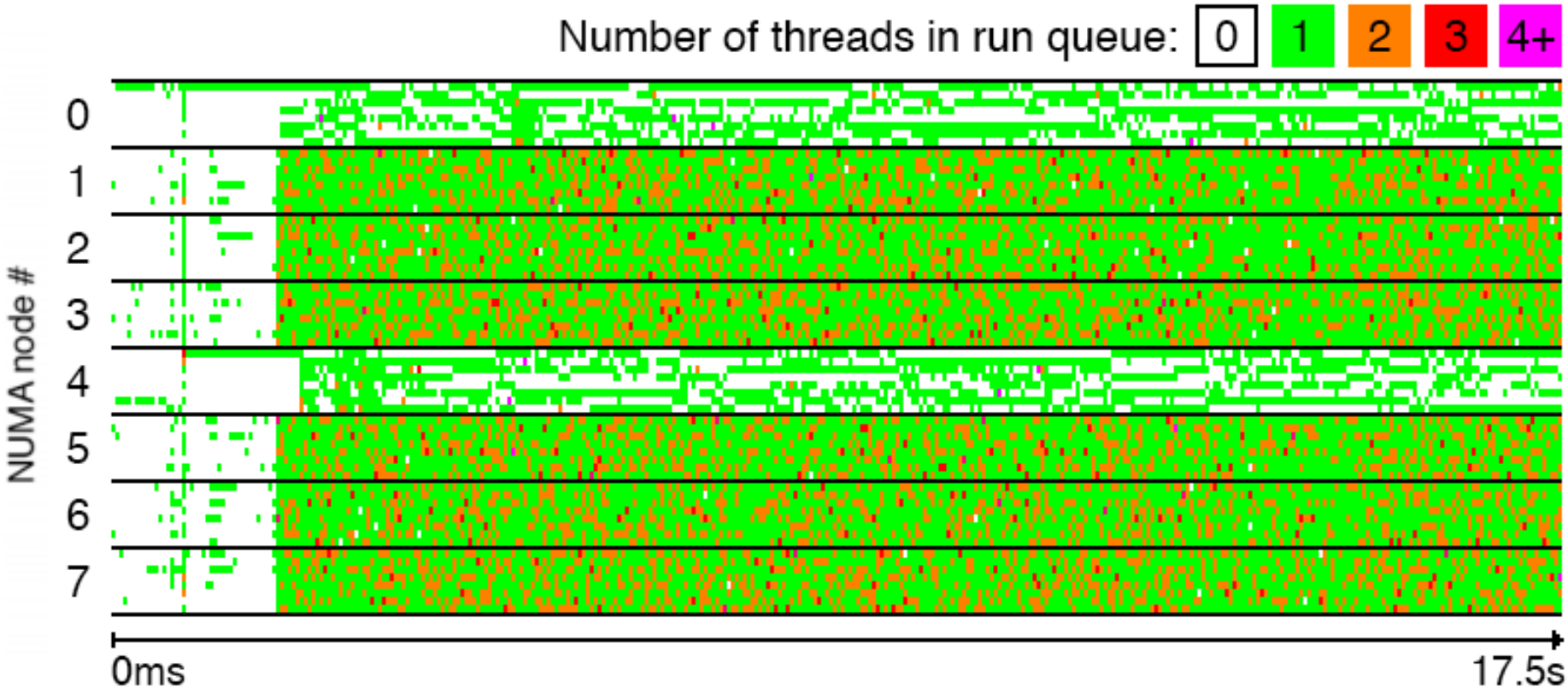
- Distribute threads on all cores of the machine.
- Deal with hardware complexities (NUMA, …)
- Handle large number of threads without overhead. Google: ~35K threads/server

Implicit contracts of a scheduler

- **Liveness**
  - All threads waiting to be scheduled will be scheduled
- **Work conservation**
  - No core should be left idle when there is work waiting to be scheduled

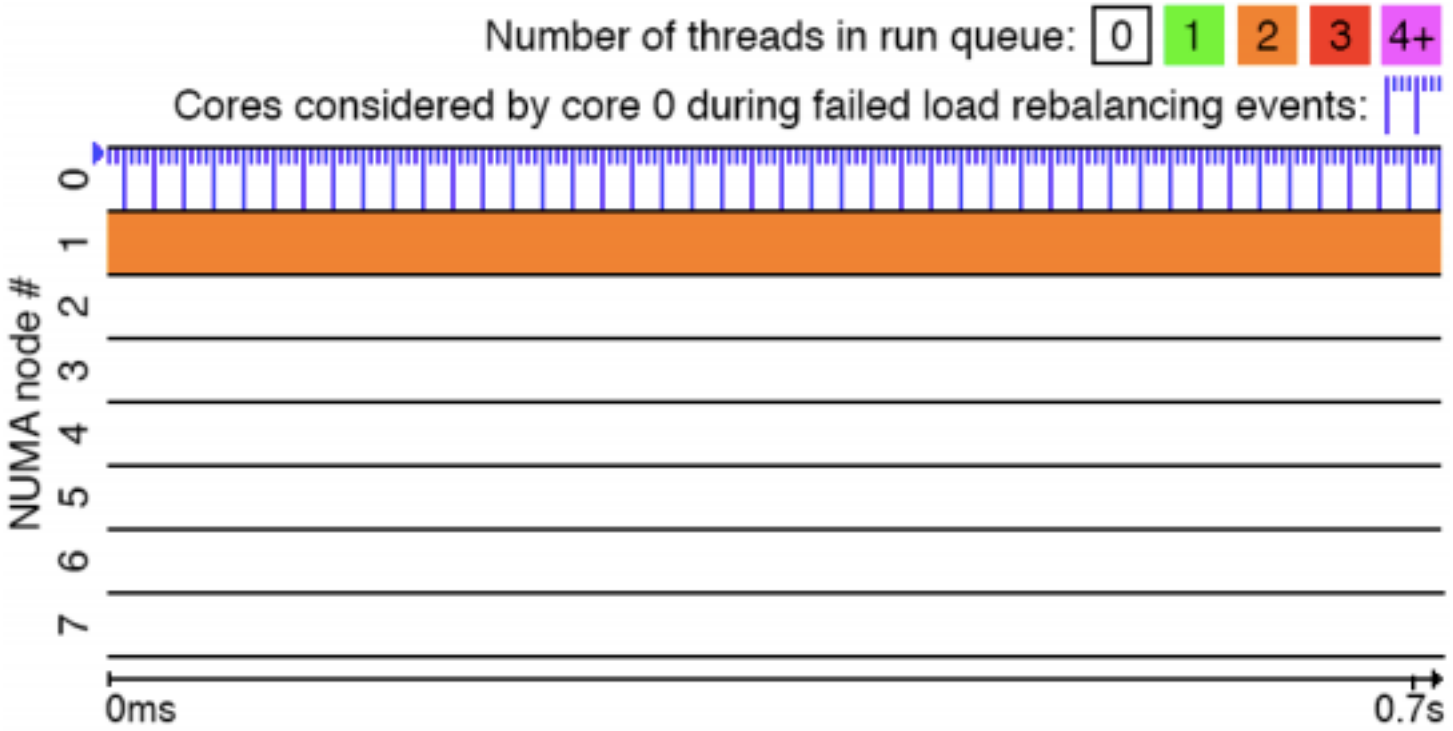
Do these contracts hold in practice?

Work conservation does not hold in Linux



R machine learning workload.
Each line is a core.
64 threads in total, 64 cores, but some cores are idle while others are overloaded

Work conservation does not hold in Linux



HPC workload.
8 cores per NUMA node.
16 threads in total, but they stay on 8 cores

This is not an intended behavior!

Linux tries to keep cores busy

Implicit contracts of a scheduler

- **Liveness**
  - All threads waiting to be scheduled will be scheduled

Work conservation (broken in Linux)

  - No core should be left idle when there is work waiting to be scheduled

Do these contracts hold in practice?

Implicit contracts of a scheduler

- **Liveness**
  - All threads waiting to be scheduled will be scheduled

Work conservation (also broken in FreeBSD!)

- No core should be left idle when there is work waiting to be scheduled

Do these contracts hold in practice?

Bug in FreeBSD:

```
void load_balance(…) {
    if(cond) // always true!
        return;
```

Load balancer actually never runs!

Implicit contracts of a scheduler

Liveness (background threads can starve in FreeBSD)

- All threads waiting to be scheduled will be scheduled

Work conservation (also broken in FreeBSD!)

- No core should be left idle when there is work waiting to be scheduled

Do these contracts hold in practice?

Proofs?

This project:

Proving work conservation in a multicore scheduler

This project

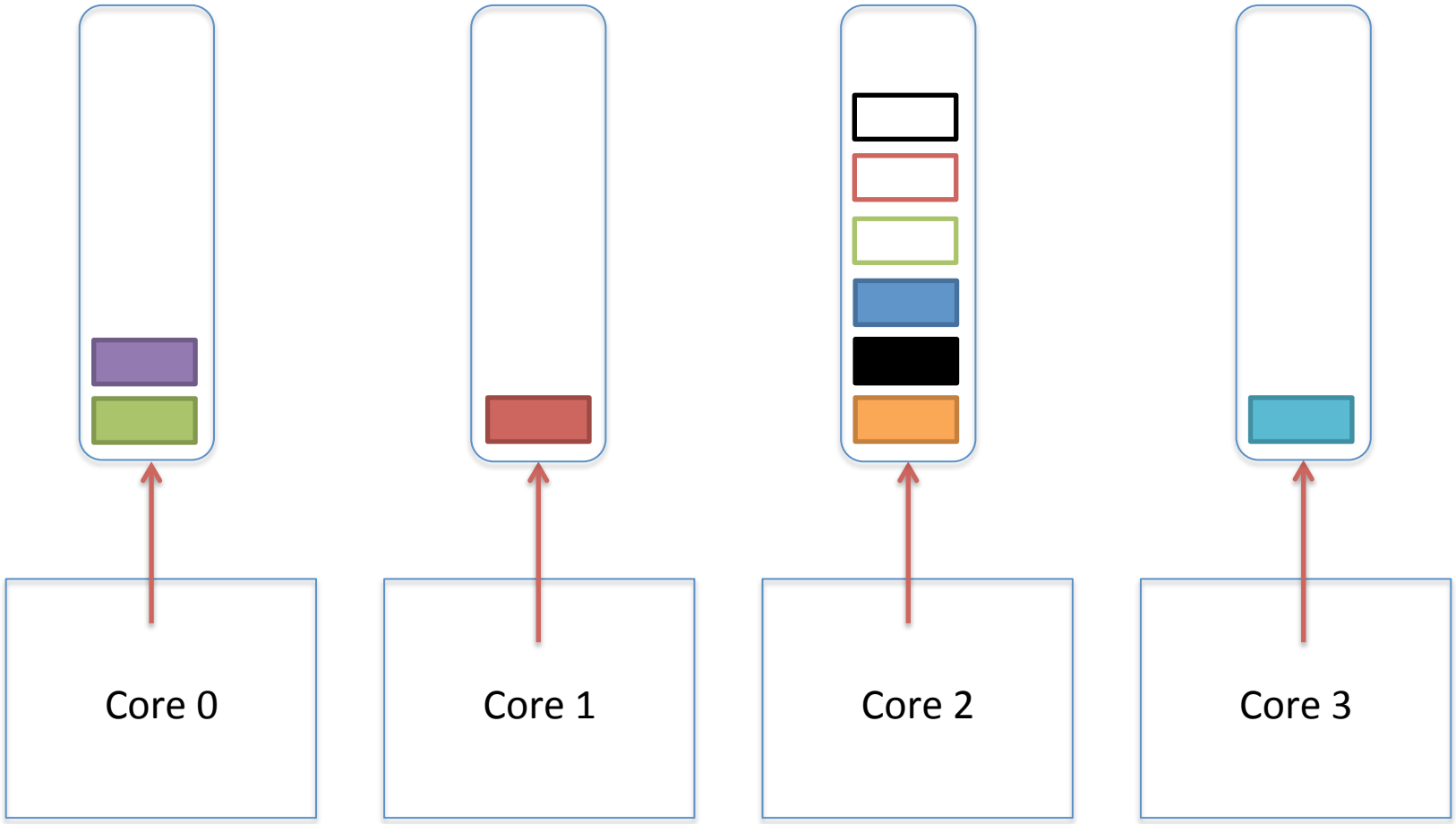
Realistic scheduler

→ Our code is inspired by CFS (default Linux scheduler)

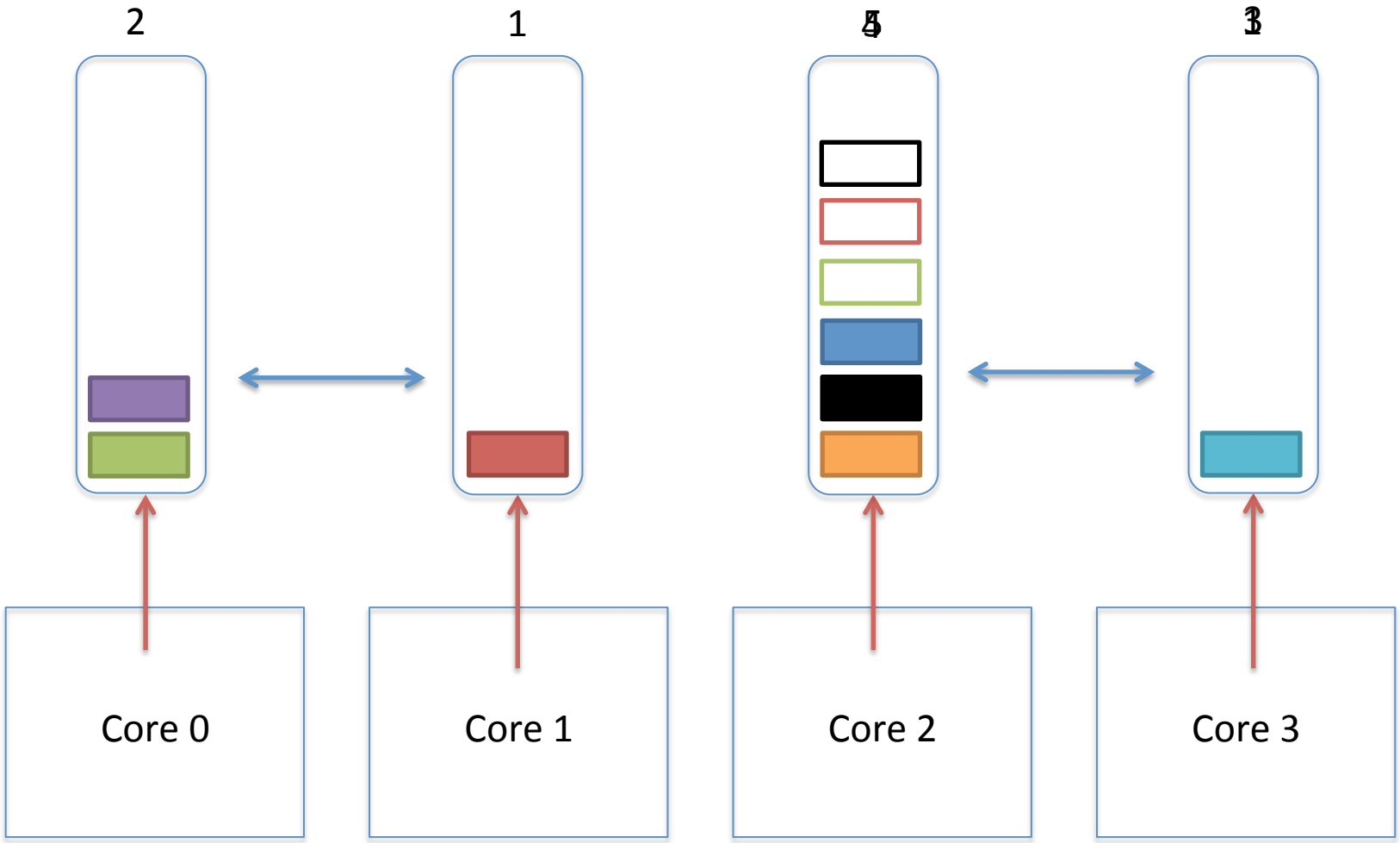
→ Must be efficient

→ Must be able to handle large number of threads

Our multicore scheduler model
1 runqueue per core



Our multicore scheduler model
Load balancing



Our multicore scheduler model
Load balancing

- Cores choose another core to steal from
- Done concurrently on all cores
- Cores may observe each others states
  - No lock!!
- No consensus: two core might decide to steal from the same 3rd core!

Challenges 1/2

- Concurrency
- Conflicts
  - Work stealing decisions use possibly out-of-date data
  - Lots of decisions taken outside of critical sections

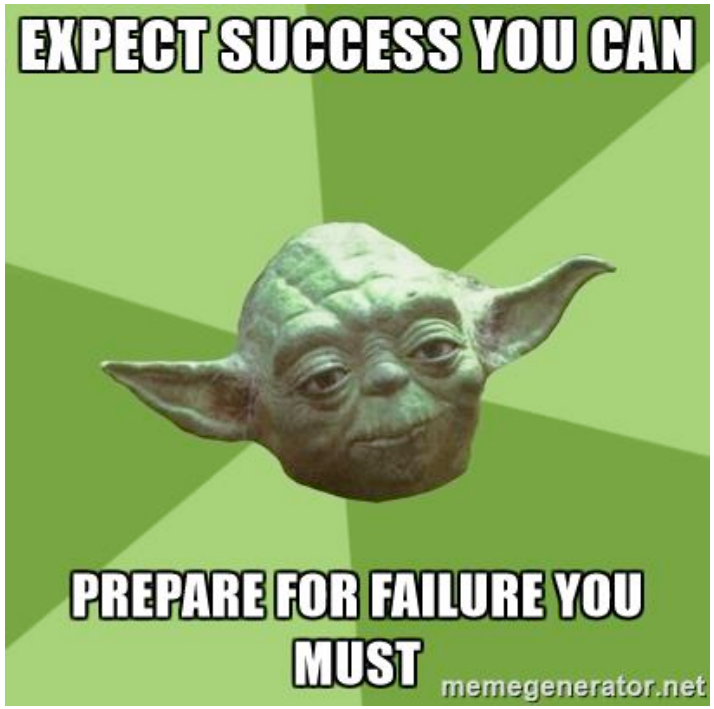
Challenges 2/2

- Work conservation doesn't "hold or break"
  - Roughly speaking:
    "Cores should not stay idle when there is work to do"
  - A core can become idle after an application exits
    - **This is not a bug**

- **Eventual property.**

Challenges (summary)

“Show that things will eventually work even though they sometimes (often?) fail to work due to conflicts”



Proofs: example

```
def load_balance(c1:Core, c2:Core) = {
   require(c1.nbthreads < c2.nbthreads)
   migrate_one_thread(c1, c2)
} ensuring(c1'.nbthreads == c1.nbthreads + 1)
```

c1' is the new state of c1

Proofs: example

```
def load_balance(c1:Core, c2:Core) = {
   require(c1.nbthreads < c2.nbthreads)
   migrate_one_thread(c1, c2)
} ensuring(c1'.nbthreads == c1.nbthreads + 1)
```

Might not be true due to
concurrent load balancing events!

Proofs: our approach

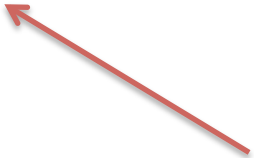
1/ Proof in a **non concurrent** case

2/ Adapt proof to **concurrent case**
Identify **invariants** that must hold
"What concurrent events can break the proof?"

3/ Check the invariants:
Only write operations may break the proof.
These write must happen a bounded amount of time

Proofs: example

```
def load_balance(c1:Core, c2:Core) = {
   require(c1.nbthreads < c2.nbthreads)
   migrate_one_thread(c1, c2)
} ensuring(c1'.nbthreads == c1.nbthreads + 1)
```



Might not be true due to concurrent load balancing events!

With a fixed workload (no thread creation/destruction), # concurrent load balancing events is bounded (after a while load is balanced on all other cores)

Proofs: example

```
def load_balance(c1:Core, c2:Core) = {
   require(c1.nbthreads < c2.nbthreads)
   migrate_one_thread(c1, c2)
} ensuring(c1'.nbthreads == c1.nbthreads + 1)
```

def global_load_imbalance = $\sum_{c1=0}^{ncores} \sum_{c2=0}^{ncores} |c1.nthreads - c2.nthreads|$

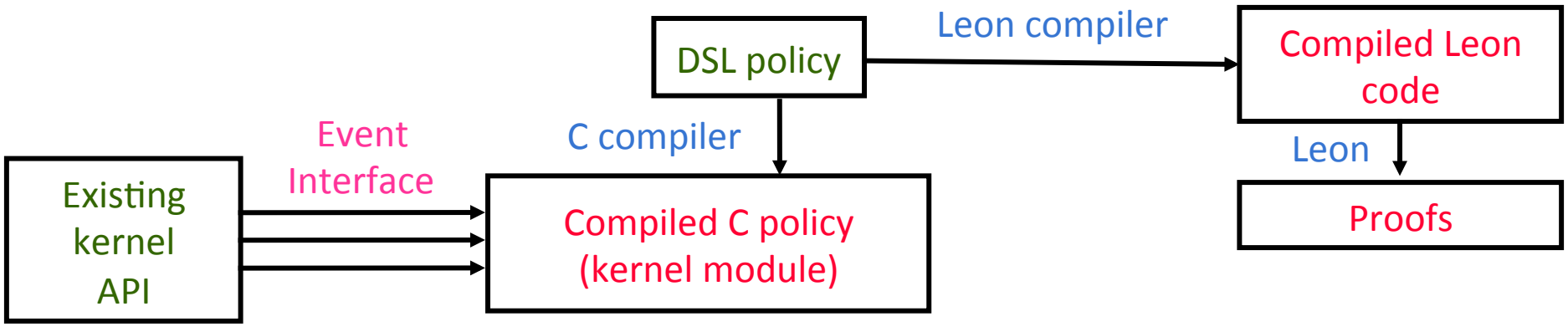
```
def progress_proof(c1:Core) = {
    decreases(global_load_imbalance) // must be true
    load_balance(c1, choose_core)        // even if that fails
     if(!load_is_balanced)
         progress_proof(c1);
}
```

Proofs: current status

Proof is done with a fixed workload
(No thread creation or destruction.)

This is already better than Linux/FreeBSD!

A word on the implementation

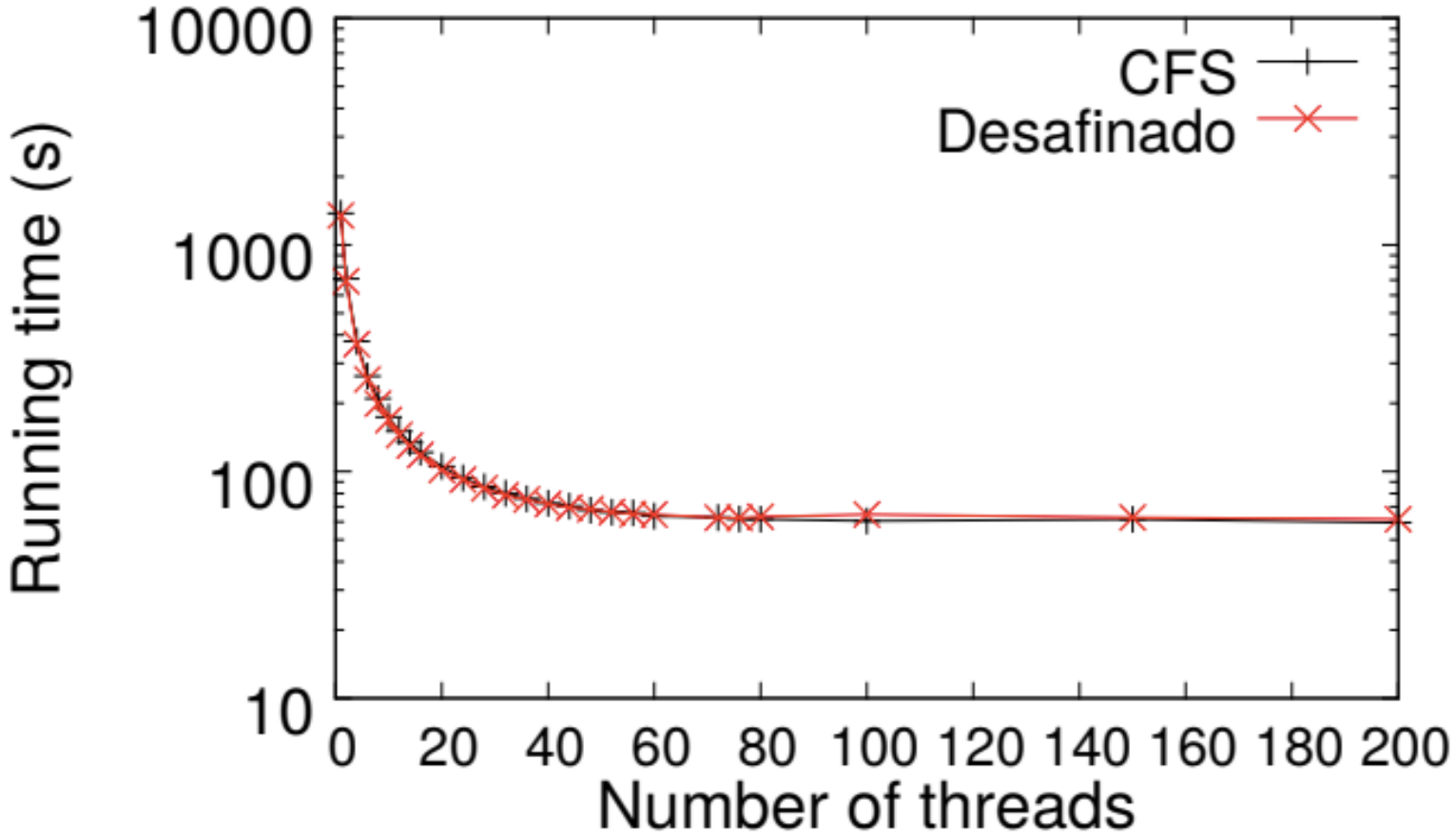


- Schedulers are written using a DSL

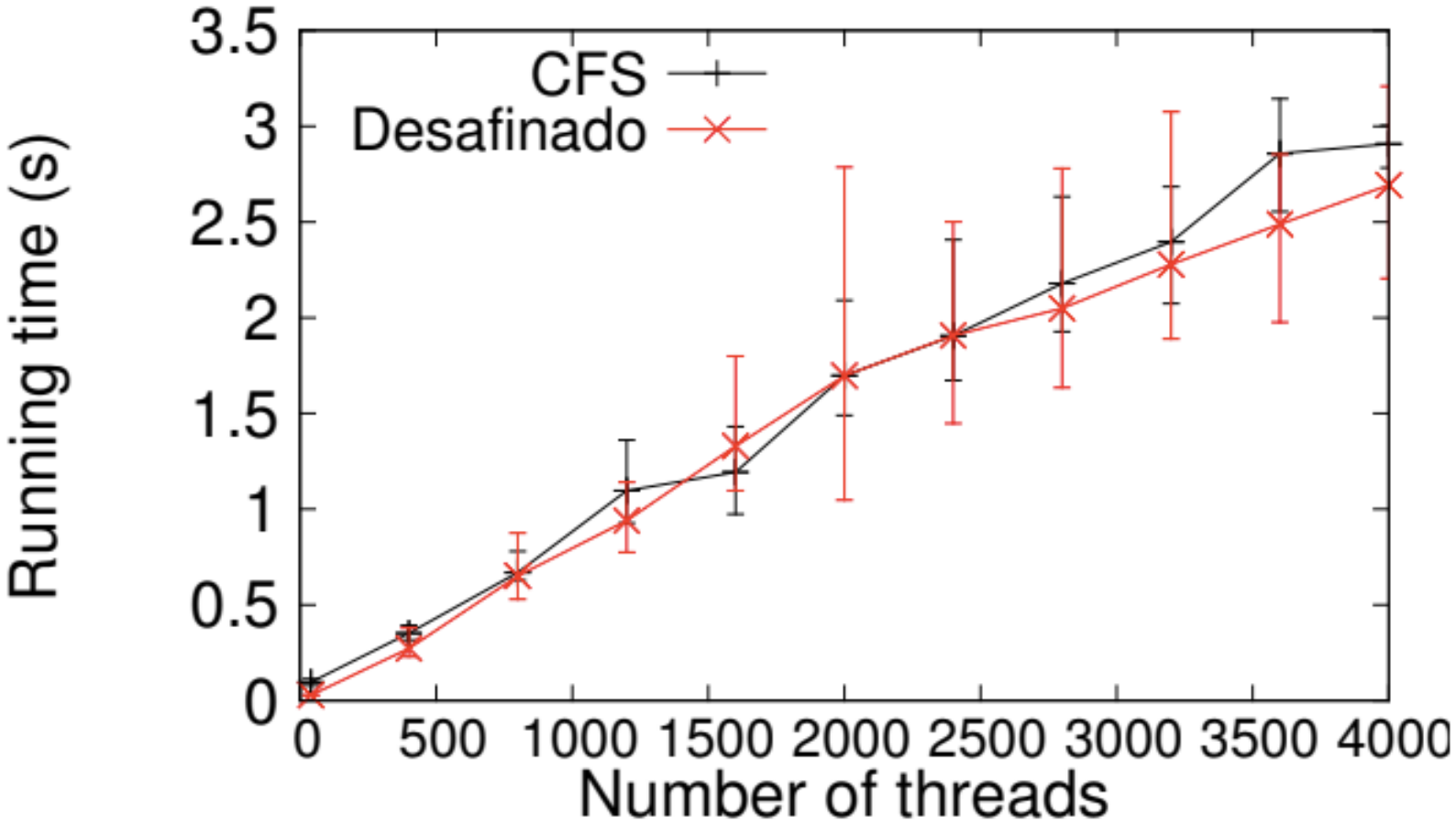
- Allows efficient compilation to C, and safety (no crash)
- Simplifies verification (DSL offers guarantees)
- Abstractions and glue between abstractions is explicit

- Proofs done in Leon (automatic verifier)

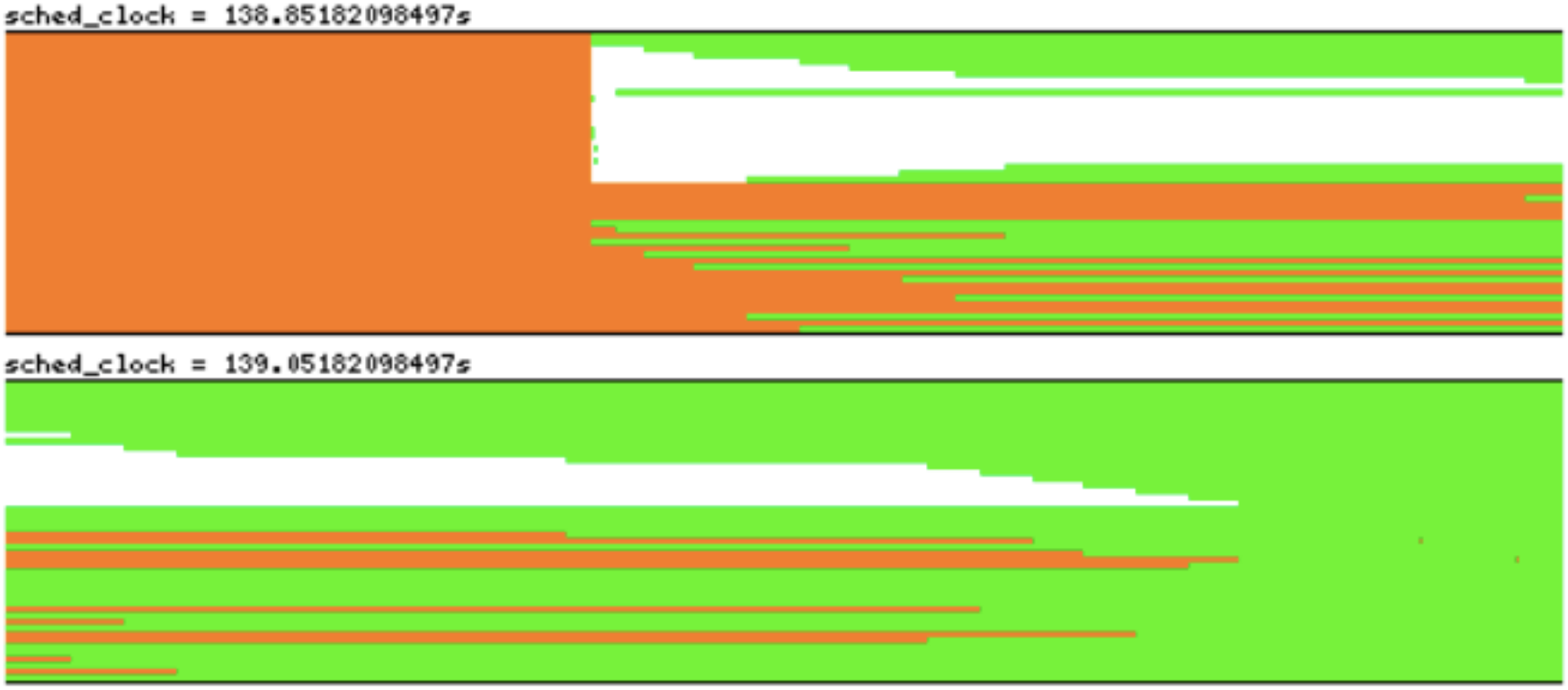
Performance - kbuild



Performance - hackbench



Performance − Time to reach balance



Orange = 2 threads, green = 1 thread
0.25 seconds (~same time as CFS)

Proofs: problem

```
def load_balance(c1:Core, c2:Core) = {
   require(c1.nbthreads < c2.nbthreads)
   migrate_one_thread(c1, c2)
} ensuring(c1'.nbthreads == c1.nbthreads + 1)


def new(p:Process) = {
    p => self.runqueue; // updates self.nbthreads
                        // breaks the proof, but shouldn't
}

def terminate(p:Process) = {
    p => self.terminated; // updates self.nbthreads
}
```

Ping pong effects!

Future work

Prove that the scheduler “tries to go in the right direction”.

- Different definition of “nbthread” that includes threads that have terminated.
- More flexible handling of what it means to “break the proof”.
- Work in progress.

Conclusion

- Schedulers are responsible of resource management.
- Work conservation does not hold in many OS.
  - Bugs in Linux & FreeBSD.
- Our project: formal verification of work conservation in realistic schedulers.
  - Hard due to high level of concurrency.
  - Proofs done in a simplified case.
  - Proofs to be done in a more realistic case.

Questions?