The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS

Justinien Bouron, Baptiste Lepers, Sébastien Chevalley, Willy Zwaenepoel
EPFL

Redha Gouicem, Julia Lawall, Gilles Muller, Julien Sopena
Sorbonne University, Inria, LIP6

What is a scheduler ?

Runs all the tasks of a system, solving the following challenges :

- Assign set of tasks to (smaller) set of cores
- High utilization of hardware resources (ie. CPU utilization)
- Fast response time and low overhead !
- React to workload changes (load balancing, ...)

Linux CFS and FreeBSD ULE

- Linux CFS is supposed to be completely fair
- FreeBSD ULE is supposed to have good interactive performances

Both schedule a large number of threads on a large number of cores ...

... but their design differ greatly

Our goal : Compare Linux CFS and FreeBSD ULE

- How do they differ in terms of design ?
- What is the impact of each design on performances ?
- Apple-to-apple comparison only, **not** declaring a winner

How to compare their impact on performances ?

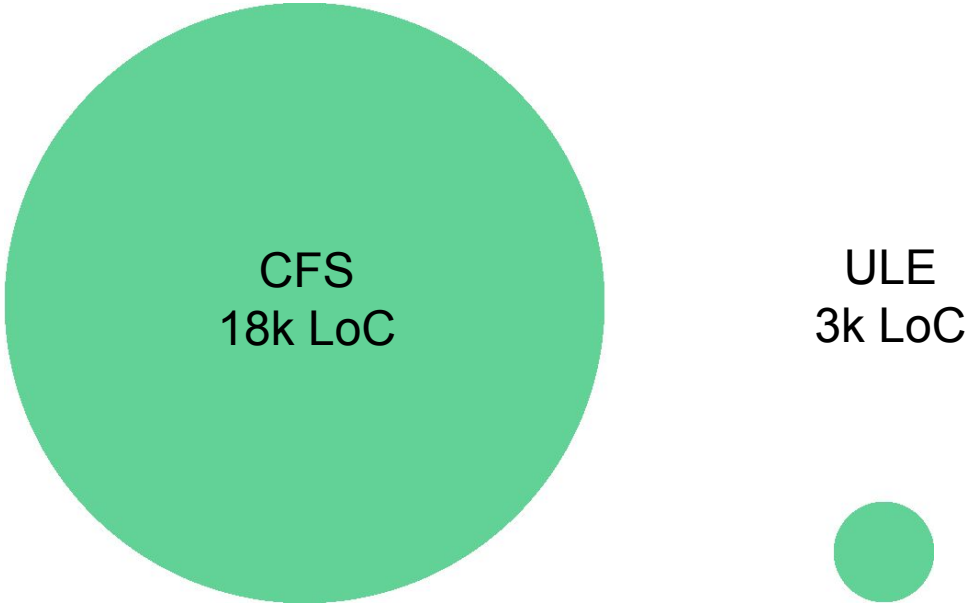
- We want to compare the impact of both scheduler on performances
- But both are from different kernels/OSes ...
- ... naively running the applications on both OSes would be highly biased

How to single out the performance differences coming from the schedulers only ?

Our approach : Transplanting

- Transplant one scheduler into the other kernel alongside the original
- Choose which one to use !
- Everything else remains the same => **No bias from other components.**

Which scheduler to transplant ?



Answer : ULE into Linux

Challenges

Challenge : Interface mismatch

- Linux provides an “API” (user defined functions) to add new schedulers ...
- ... FreeBSD does not
- But functions inside ULE could easily be mapped to their Linux counterpart

Challenge : Different low-level assumptions

Both schedulers have their very own low-level assumptions :

- Locking policy : Multiple locks
- Runqueue management : data structures, locks, indexes, ...
- Priority range : CFS is nice range, ULE all tasks

ULE’s code had to be slightly modified to comply with Linux’s assumptions

Evaluation

A broader performance comparison

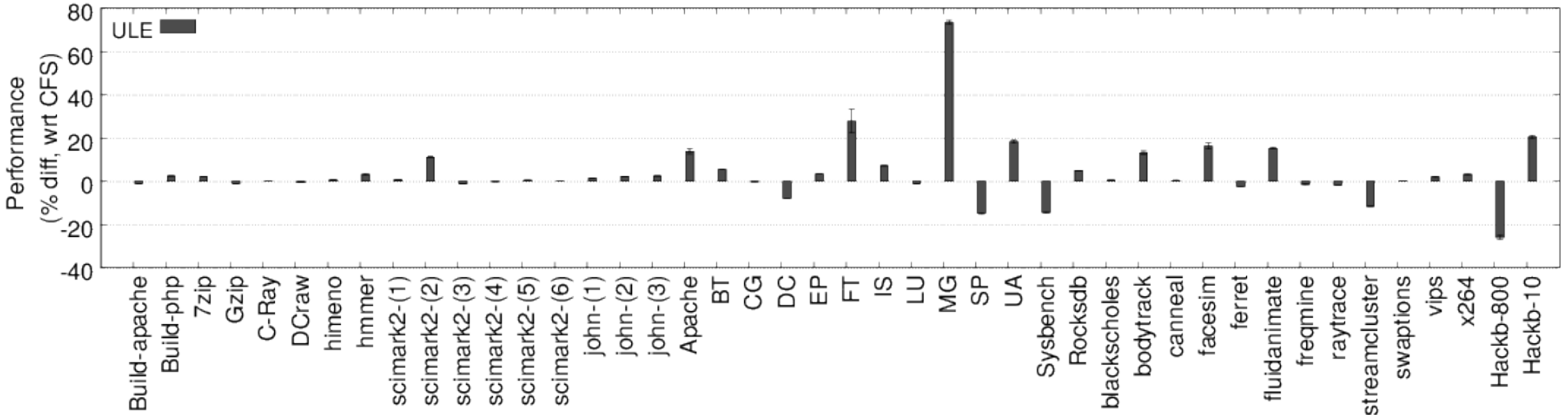
- Mix of synthetic benchmarks and realistic applications
- Evaluation performed on a 32-cores NUMA machine (AMD Opteron) with 32GB of RAM.

Dual-purpose :

- Test our implementation
- Give us clues on where to look for differences

A broader performance comparison

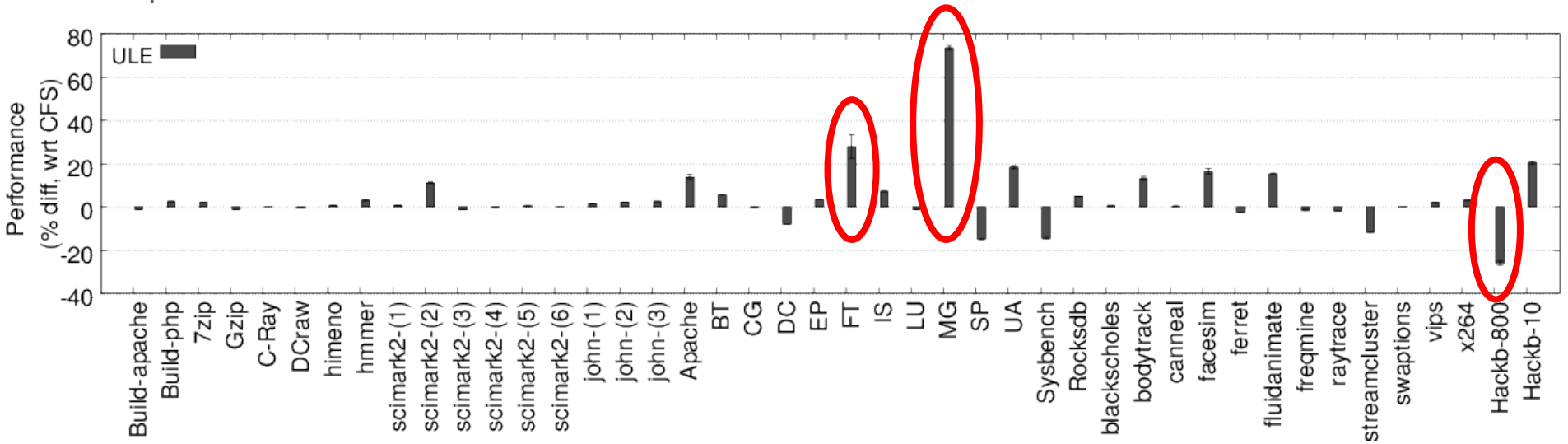
- Most of the time the performances are the same : 2.75% in favor of ULE in average.



Performances of ULE (% diff w.r.t CFS)

A broader performance comparison

- Big gaps when changing scheduler : The scheduler can have a big impact on performances !



Performances of ULE (% diff w.r.t CFS)

Design differences

Difference #1/4 :
Dealing with interactive tasks

Dealing with interactive tasks : The difference

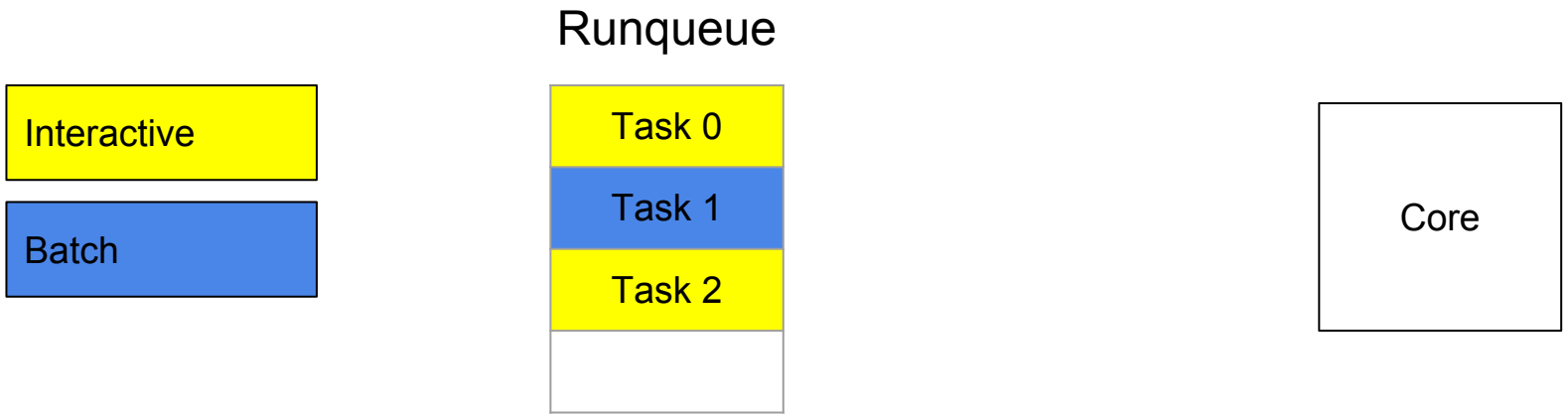
- A task can be either *interactive* or *batch*
- *Interactive* tasks = sleep most of the time (inputs, yields, ...)
- *Batch* tasks = CPU-bound tasks with very little sleep

How are CFS and ULE handling those ?

CFS

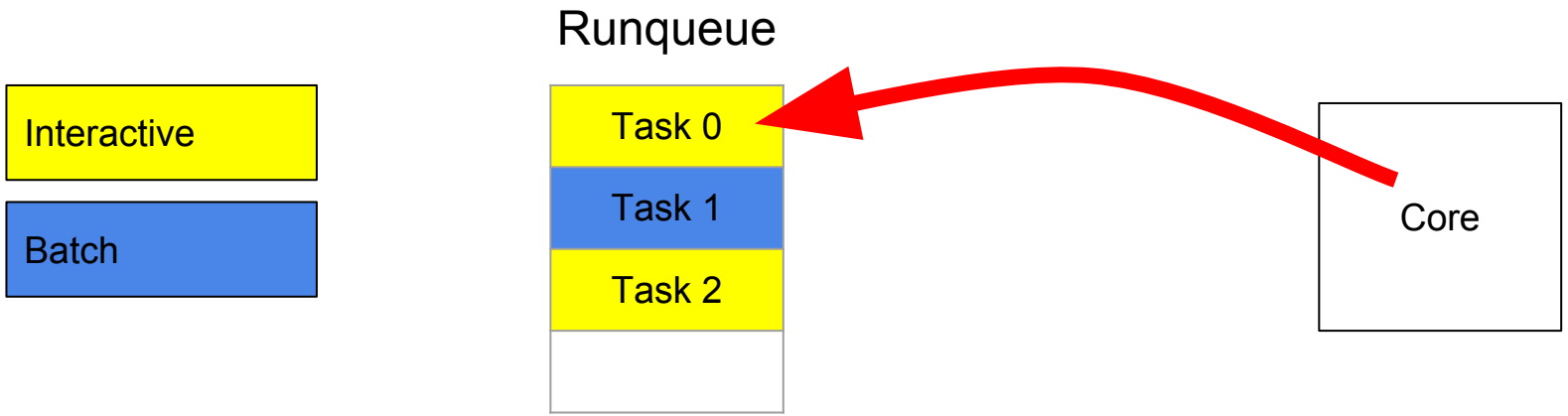
Dealing with interactive tasks : CFS

- CFS is fair, thus no distinction between interactive and batch tasks.
- Tasks ordered by runtime, pick the one on top of the runqueue



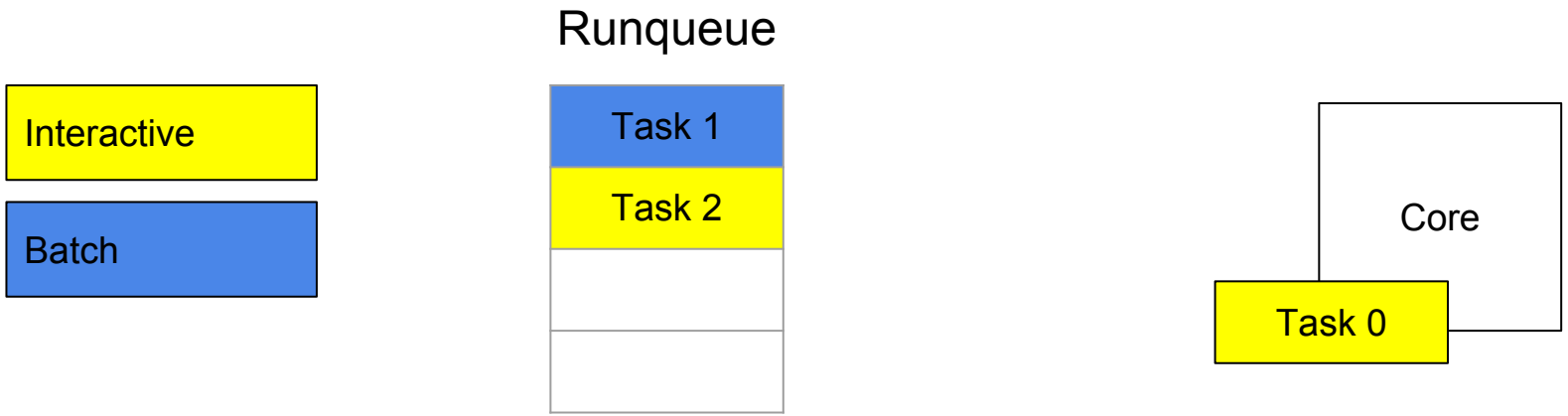
Dealing with interactive tasks : CFS

- CFS is fair, thus no distinction between interactive and batch tasks.
- Tasks ordered by runtime, pick the one on top of the runqueue



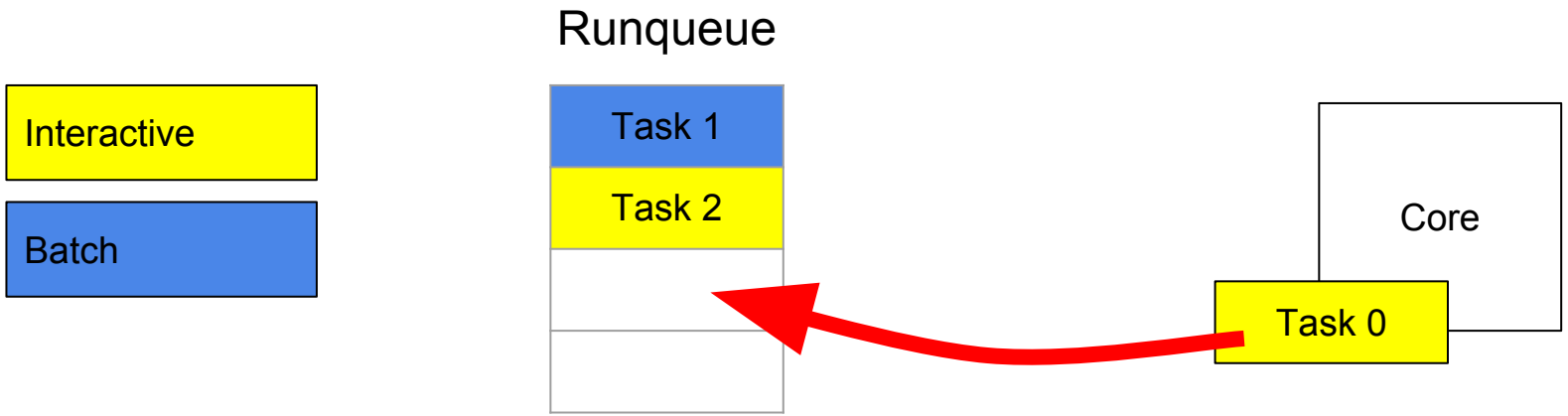
Dealing with interactive tasks : CFS

- CFS is fair, thus no distinction between interactive and batch tasks.
- Tasks ordered by runtime, pick the one on top of the runqueue



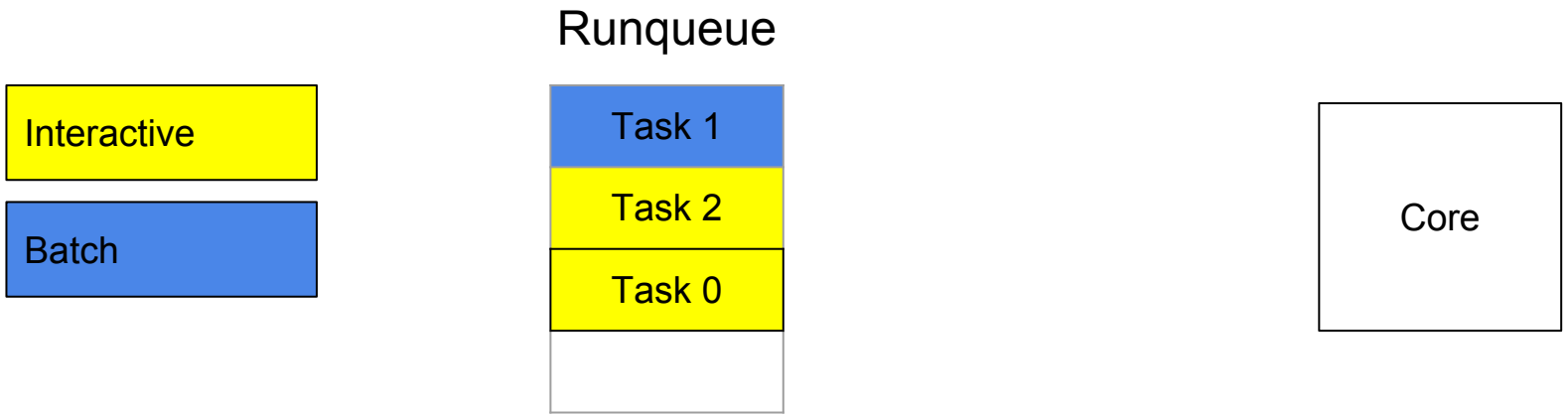
Dealing with interactive tasks : CFS

- CFS is fair, thus no distinction between interactive and batch tasks.
- Tasks ordered by runtime, pick the one on top of the runqueue



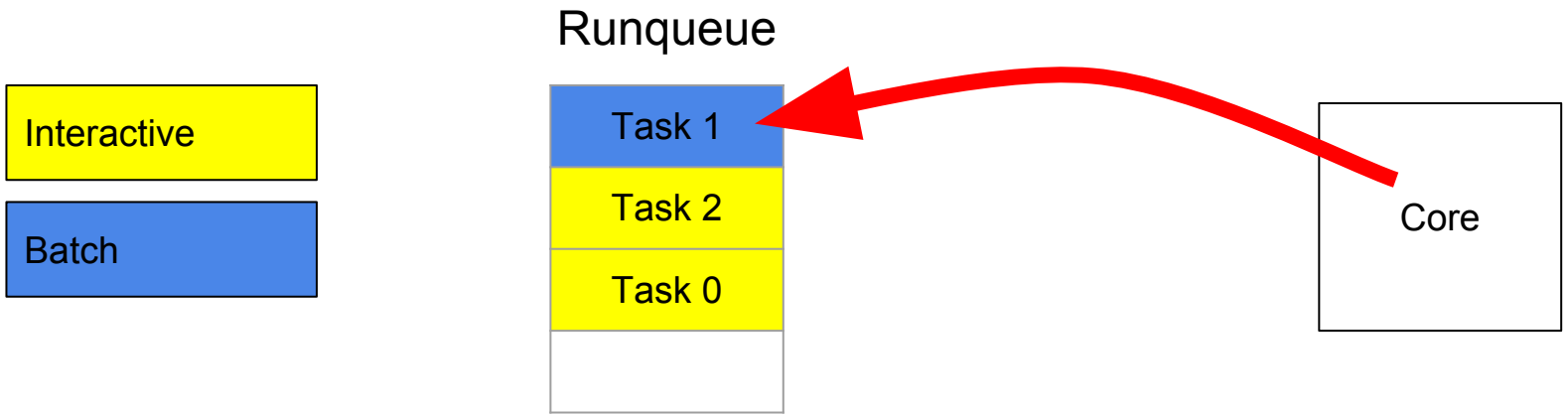
Dealing with interactive tasks : CFS

- CFS is fair, thus no distinction between interactive and batch tasks.
- Tasks ordered by runtime, pick the one on top of the runqueue



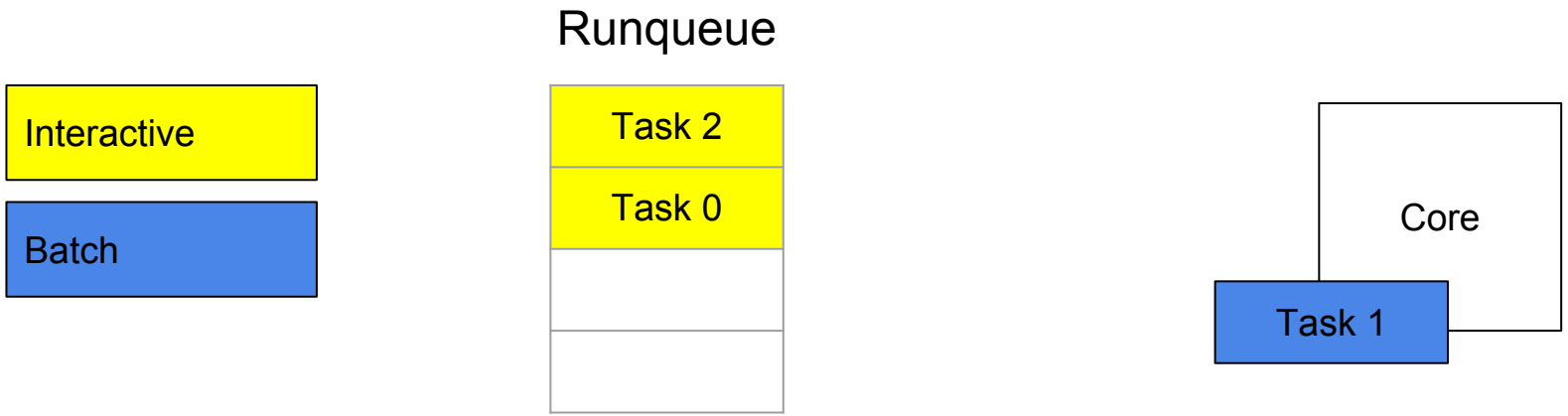
Dealing with interactive tasks : CFS

- CFS is fair, thus no distinction between interactive and batch tasks.
- Tasks ordered by runtime, pick the one on top of the runqueue



Dealing with interactive tasks : CFS

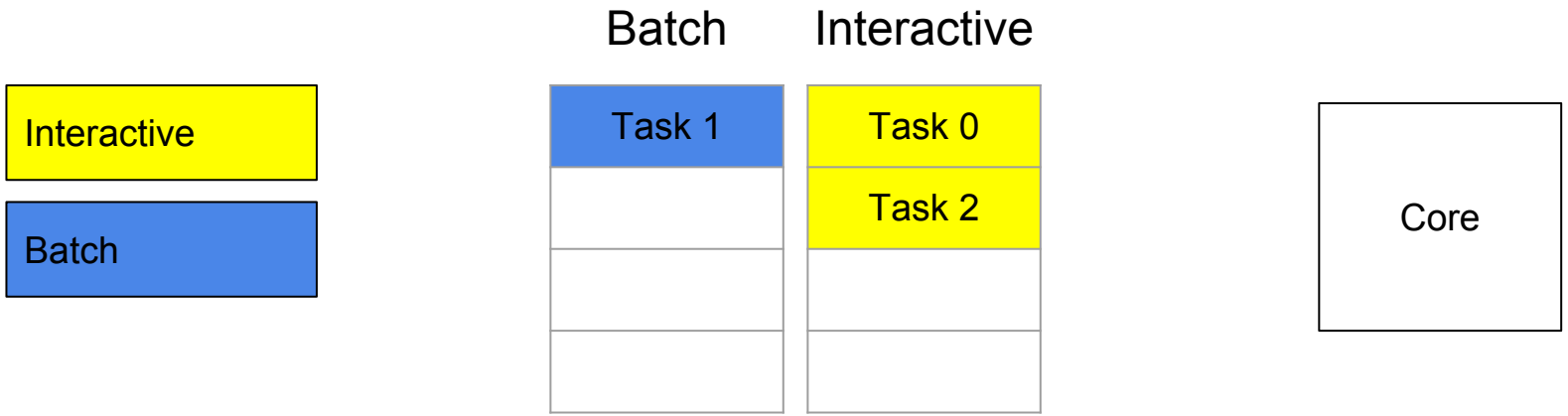
- CFS is fair, thus no distinction between interactive and batch tasks.
- Tasks ordered by runtime, pick the one on top of the runqueue



ULE

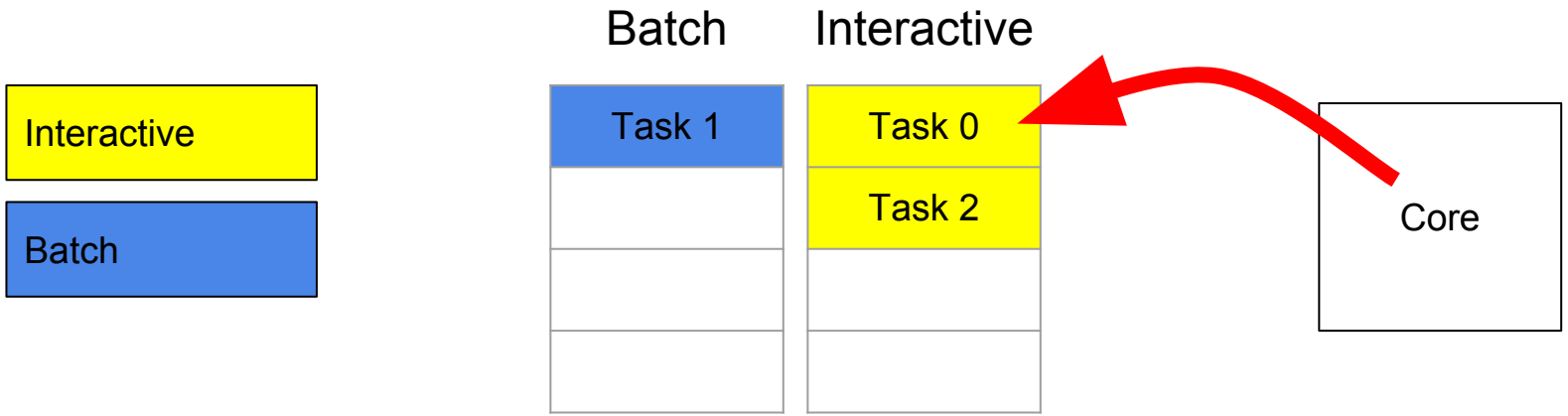
Dealing with interactive tasks : ULE

- ULE keeps interactive tasks and batch tasks in separate runqueues
- Tasks ordered by runtime in each
- Interactive tasks have absolute priority over batch tasks



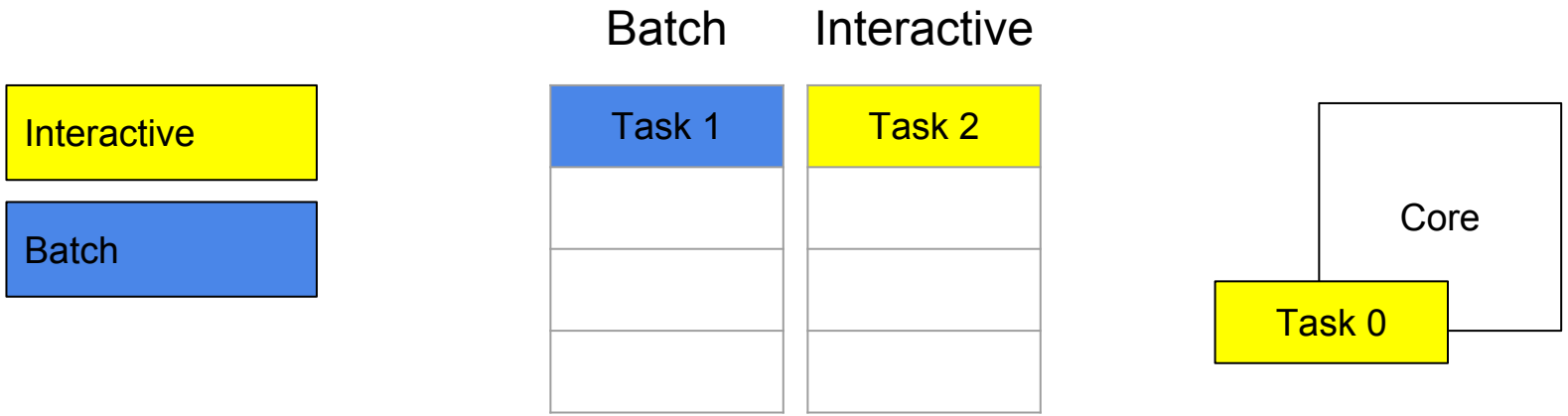
Dealing with interactive tasks : ULE

- ULE keeps interactive tasks and batch tasks in separate runqueues
- Tasks ordered by runtime in each
- Interactive tasks have absolute priority over batch tasks



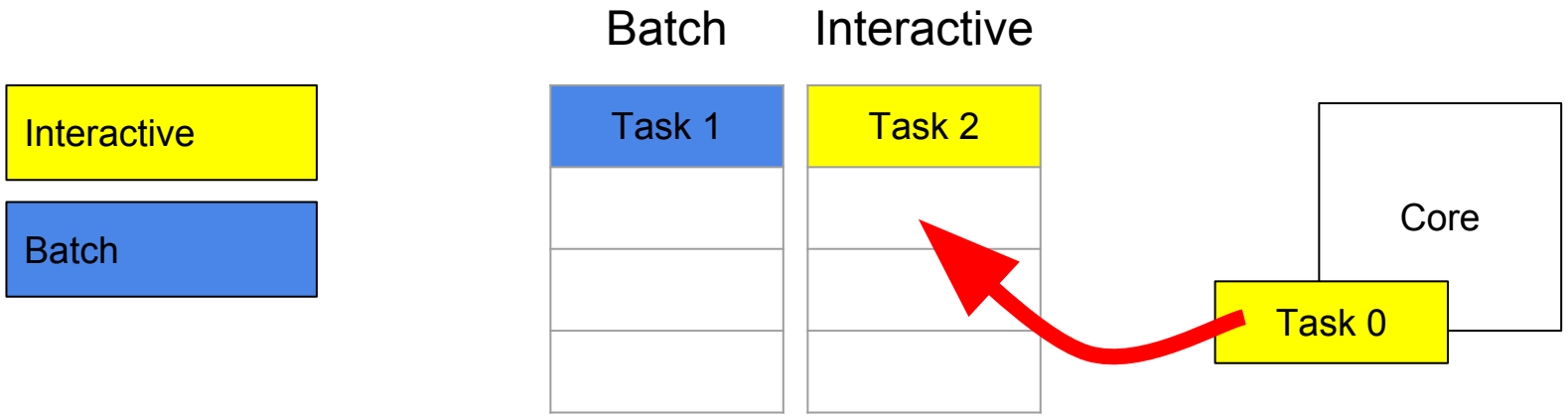
Dealing with interactive tasks : ULE

- ULE keeps interactive tasks and batch tasks in separate runqueues
- Tasks ordered by runtime in each
- Interactive tasks have absolute priority over batch tasks



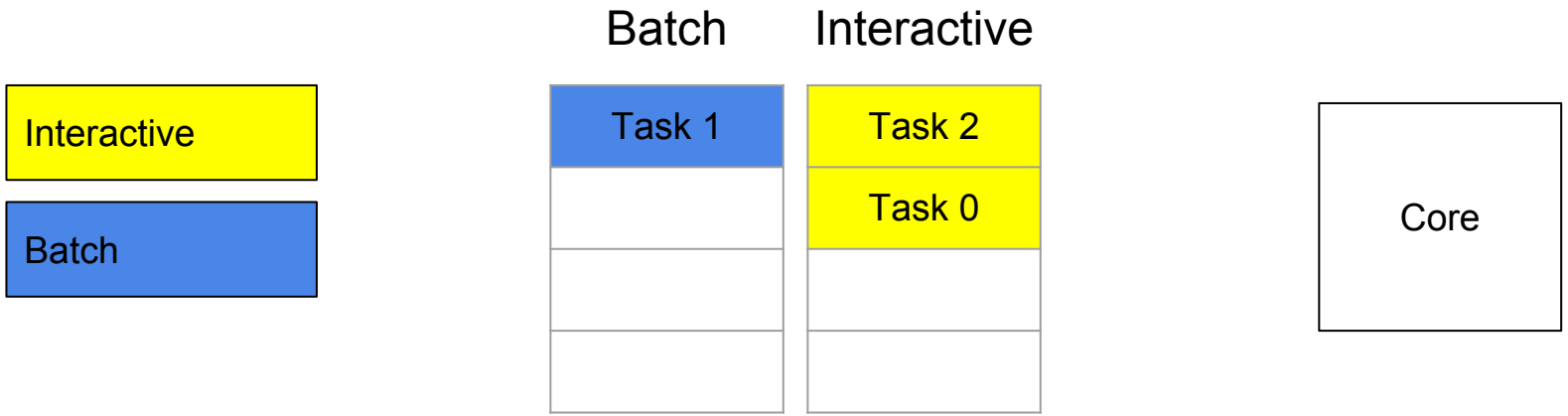
Dealing with interactive tasks : ULE

- ULE keeps interactive tasks and batch tasks in separate runqueues
- Tasks ordered by runtime in each
- Interactive tasks have absolute priority over batch tasks



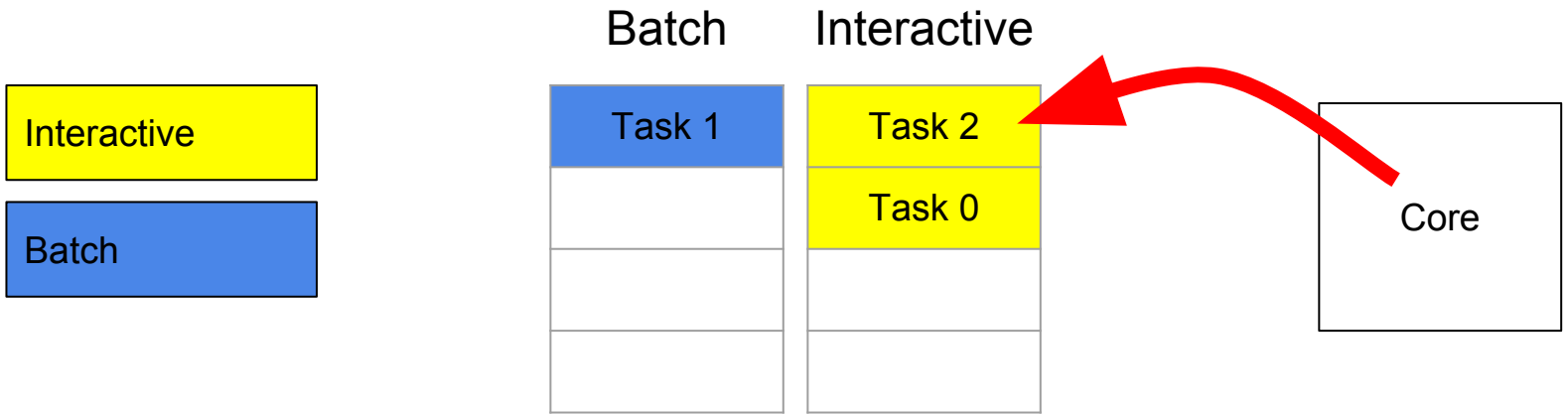
Dealing with interactive tasks : ULE

- ULE keeps interactive tasks and batch tasks in separate runqueues
- Tasks ordered by runtime in each
- Interactive tasks have absolute priority over batch tasks



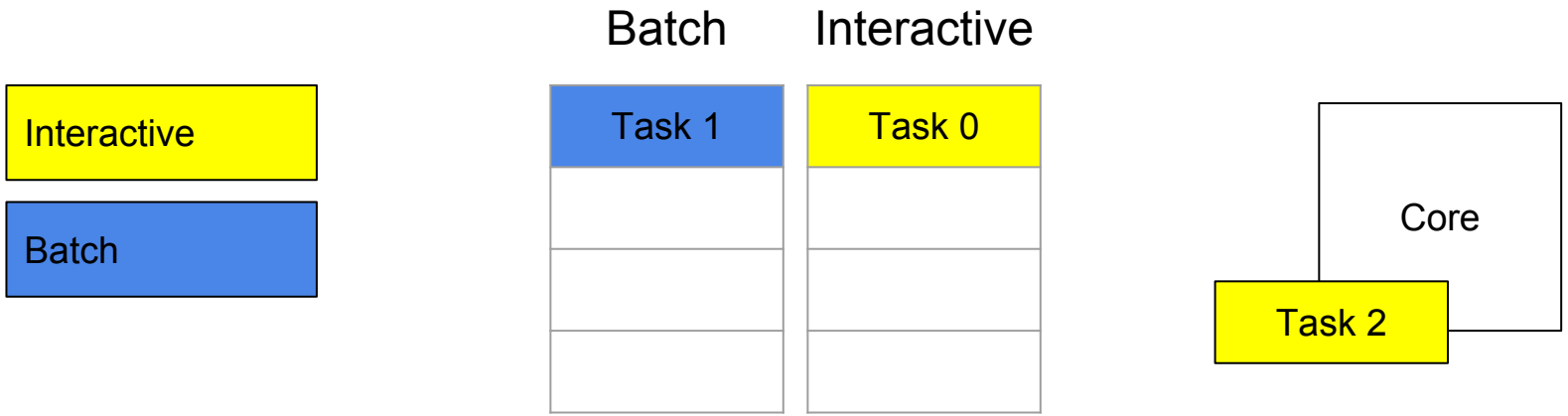
Dealing with interactive tasks : ULE

- ULE keeps interactive tasks and batch tasks in separate runqueues
- Tasks ordered by runtime in each
- Interactive tasks have absolute priority over batch tasks



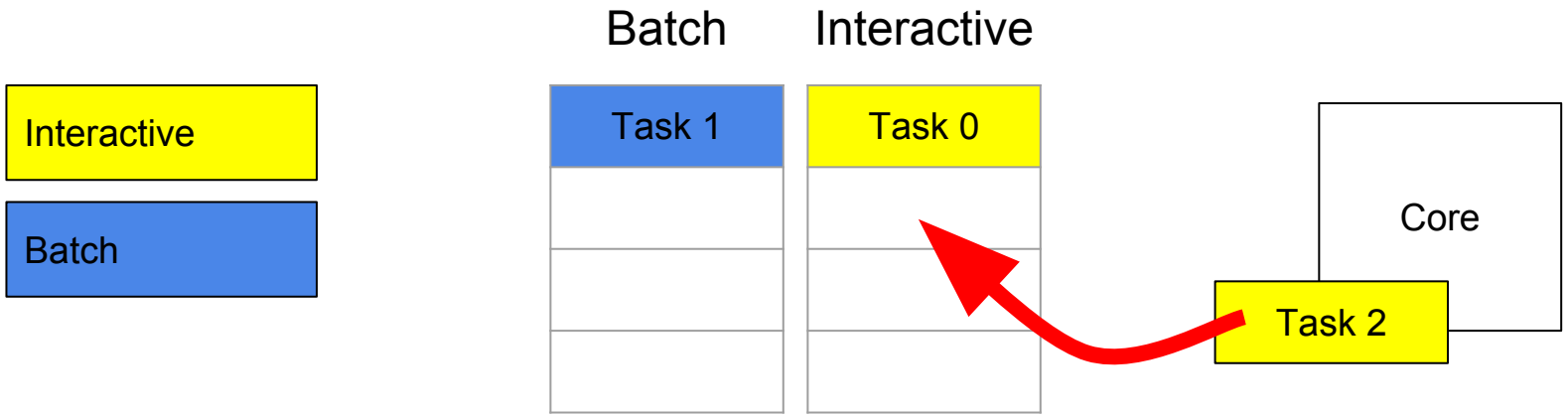
Dealing with interactive tasks : ULE

- ULE keeps interactive tasks and batch tasks in separate runqueues
- Tasks ordered by runtime in each
- Interactive tasks have absolute priority over batch tasks



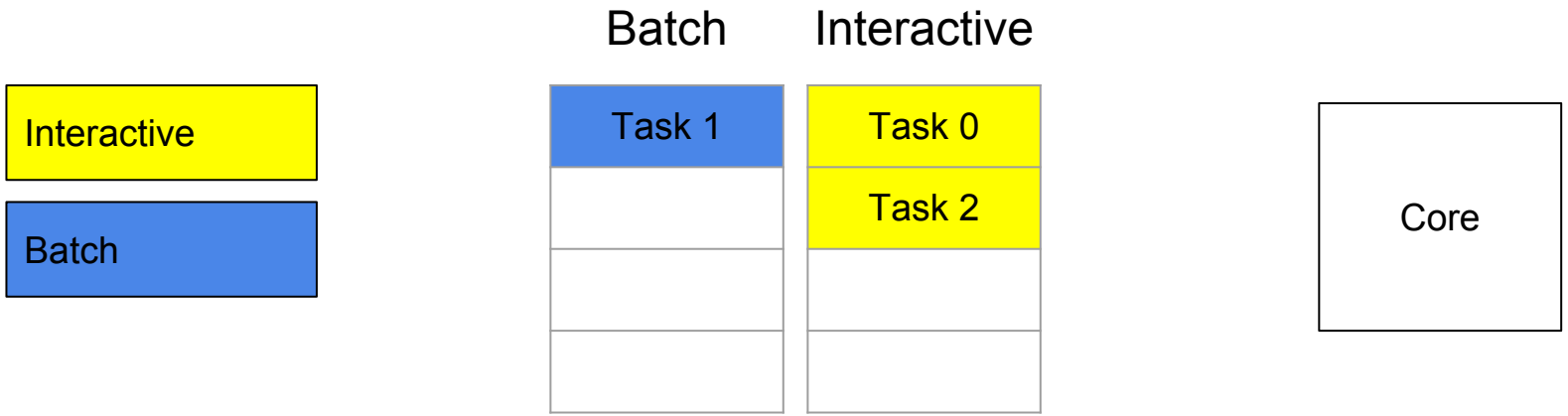
Dealing with interactive tasks : ULE

- ULE keeps interactive tasks and batch tasks in separate runqueues
- Tasks ordered by runtime in each
- Interactive tasks have absolute priority over batch tasks



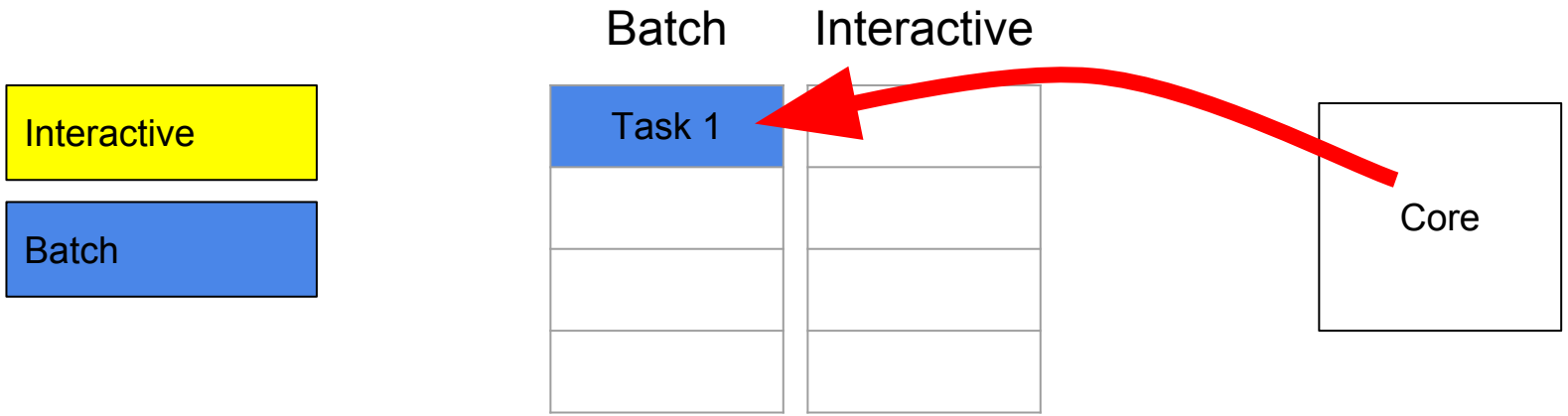
Dealing with interactive tasks : ULE

- ULE keeps interactive tasks and batch tasks in separate runqueues
- Tasks ordered by runtime in each
- Interactive tasks have absolute priority over batch tasks



Dealing with interactive tasks : ULE

- ULE keeps interactive tasks and batch tasks in separate runqueues
- Tasks ordered by runtime in each
- Interactive tasks have absolute priority over batch tasks

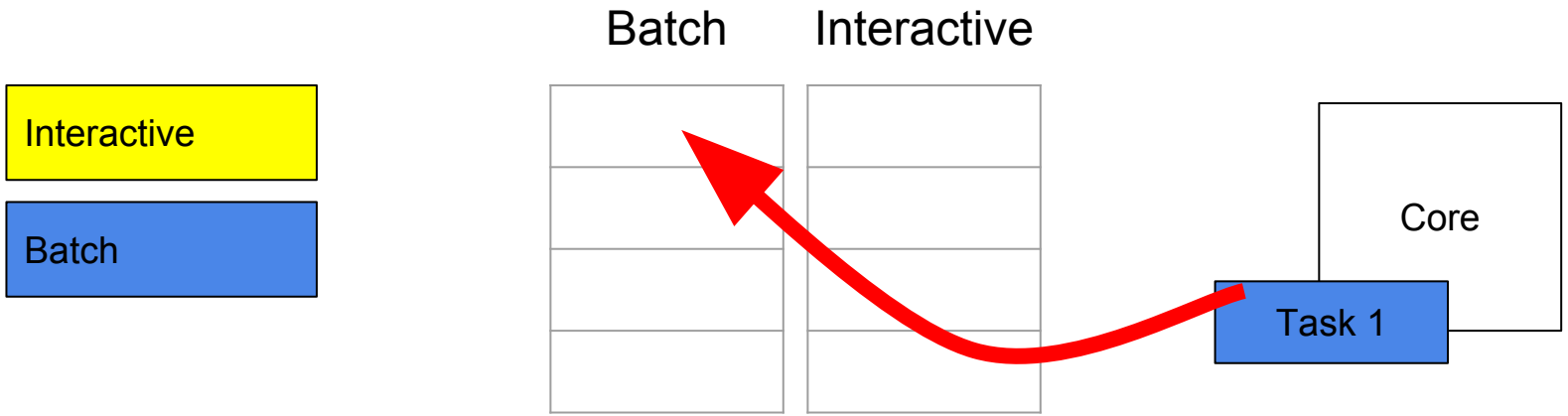


Dealing with interactive tasks : ULE

- ULE keeps interactive tasks and batch tasks in separate runqueues
- Tasks ordered by runtime in each
- Interactive tasks have absolute priority over batch tasks

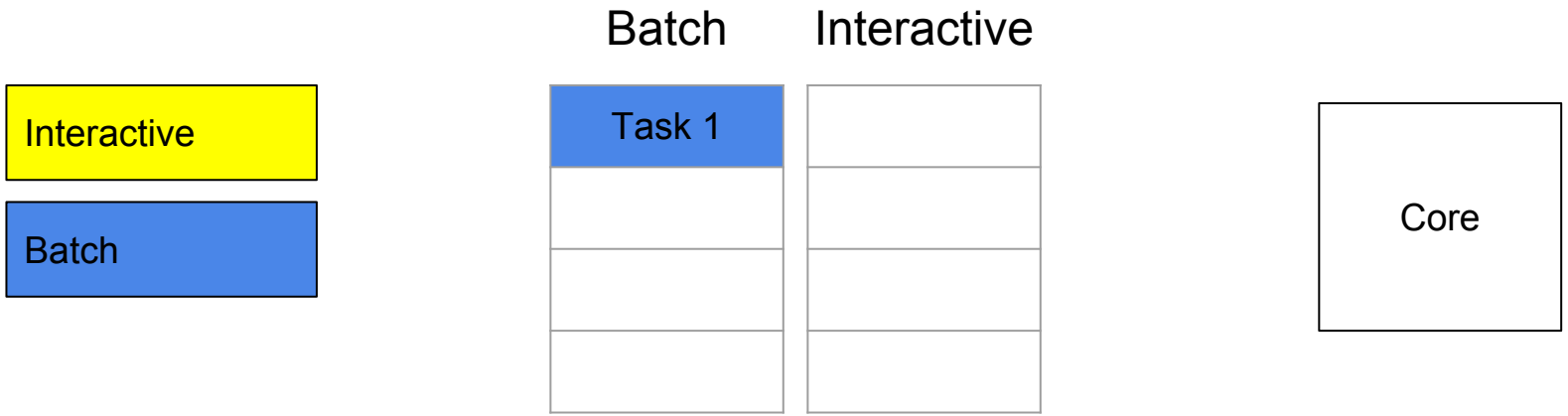
Dealing with interactive tasks : ULE

- ULE keeps interactive tasks and batch tasks in separate runqueues
- Tasks ordered by runtime in each
- Interactive tasks have absolute priority over batch tasks



Dealing with interactive tasks : ULE

- ULE keeps interactive tasks and batch tasks in separate runqueues
- Tasks ordered by runtime in each
- Interactive tasks have absolute priority over batch tasks



Dealing with interactive tasks : Rationales

CFS’s rationale :

- Let’s be fair
- No distinction between tasks

ULE’s rationale :

- Interactive tasks are latency-critical, give them absolute priority
- This should not cause problems as they sleep most of the time

Dealing with interactive tasks : Similarities

Both schedulers operate the same way when dealing with only one class of task

- They both pick the task with the lowest runtime from one runqueue

Thus the only interesting case to study is when we mix both classes of tasks

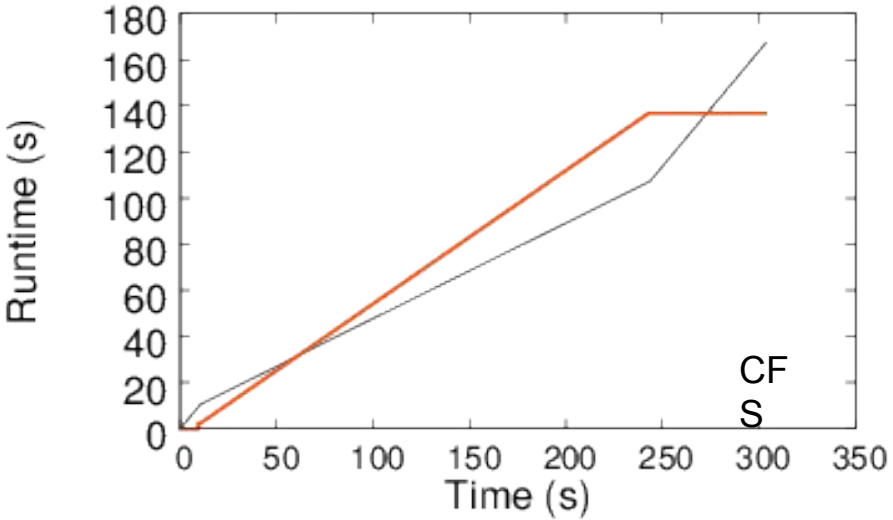
Dealing with interactive tasks : The experiment

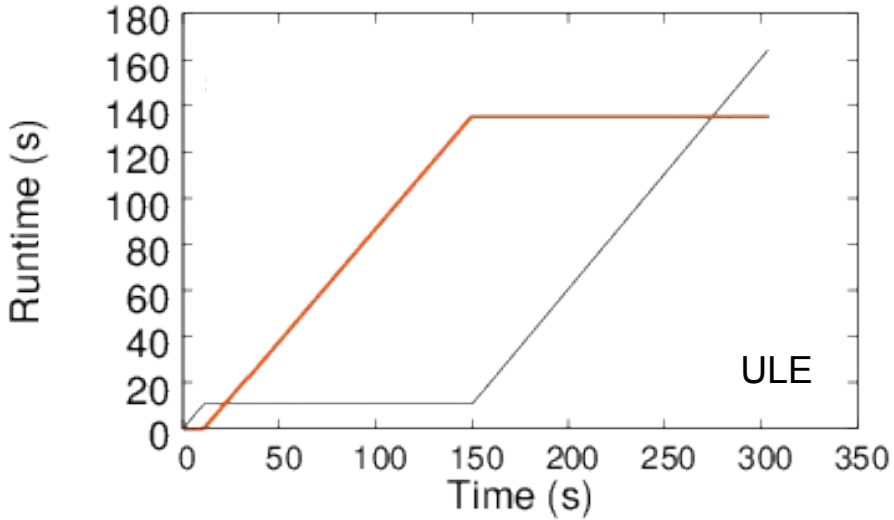
Run two applications in parallel on a single core machine ...

- One interactive application with 80 interactive threads
- One single-threaded batch application

Goal : compare the evolution of their runtime

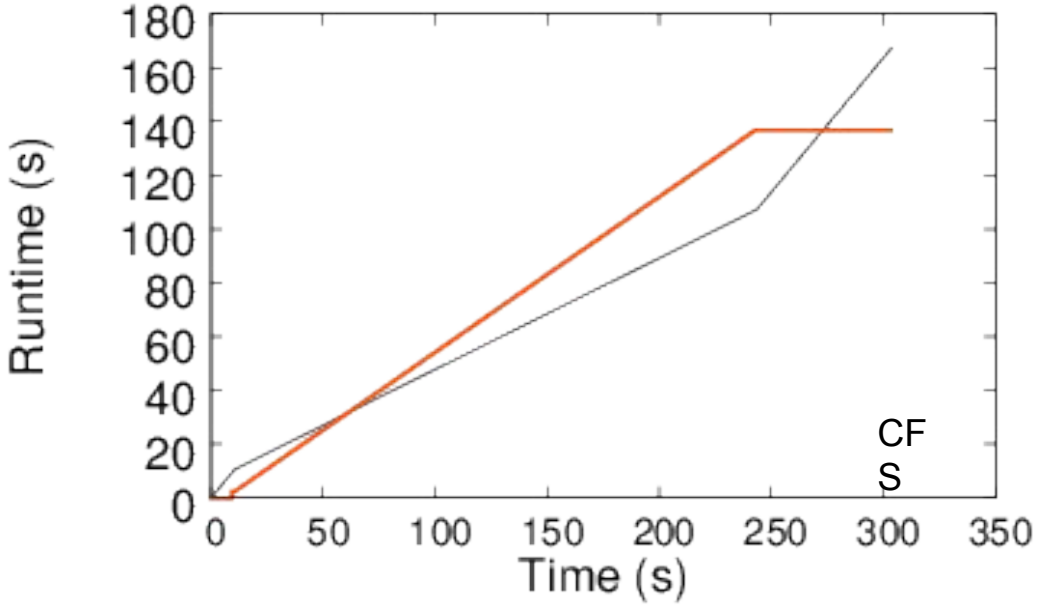
Dealing with interactive tasks : The impact



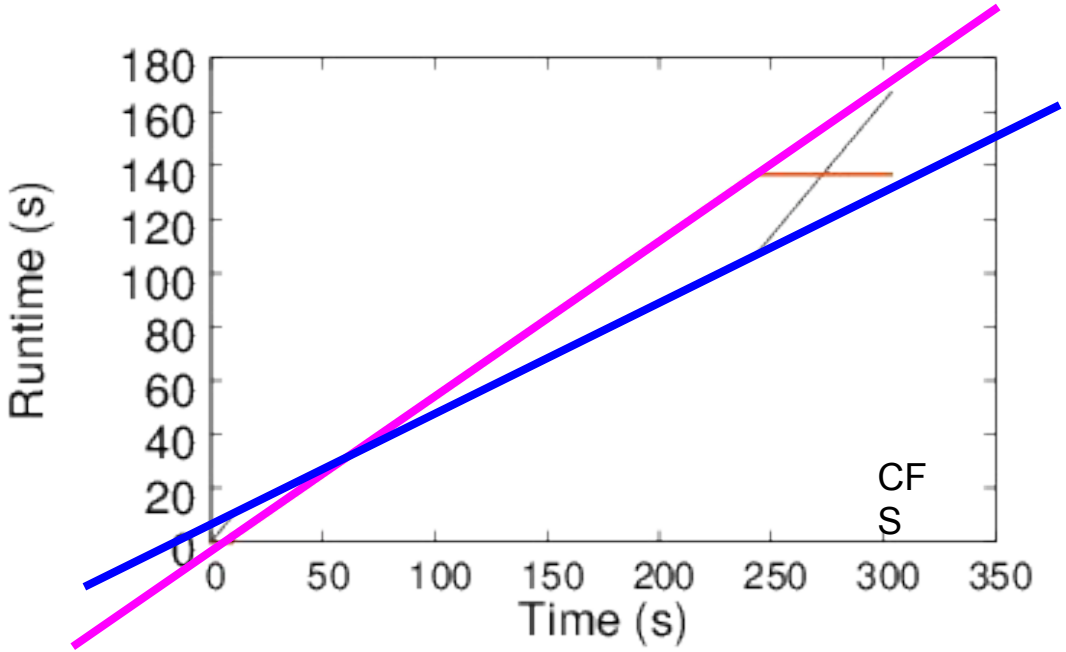


Interactive Batch

Dealing with interactive tasks : The impact



Dealing with interactive tasks : The impact

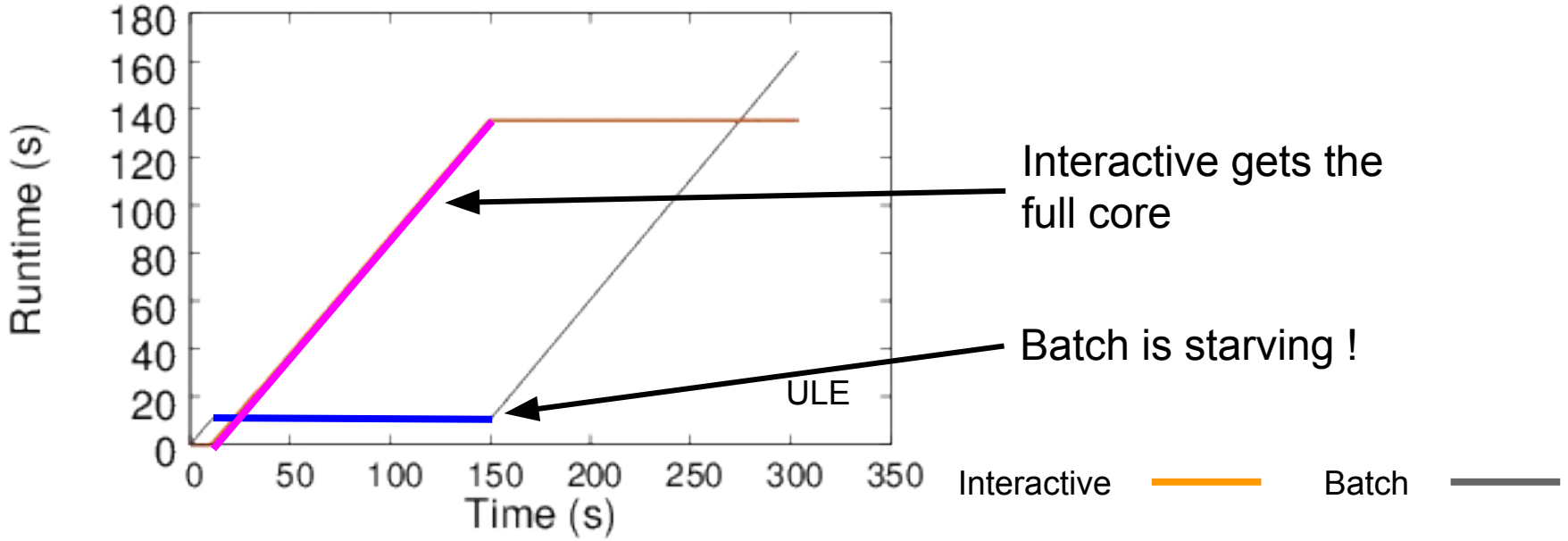


Roughly same slope for
Interactive and Batch
=> Fairness

Interactive — Batch —

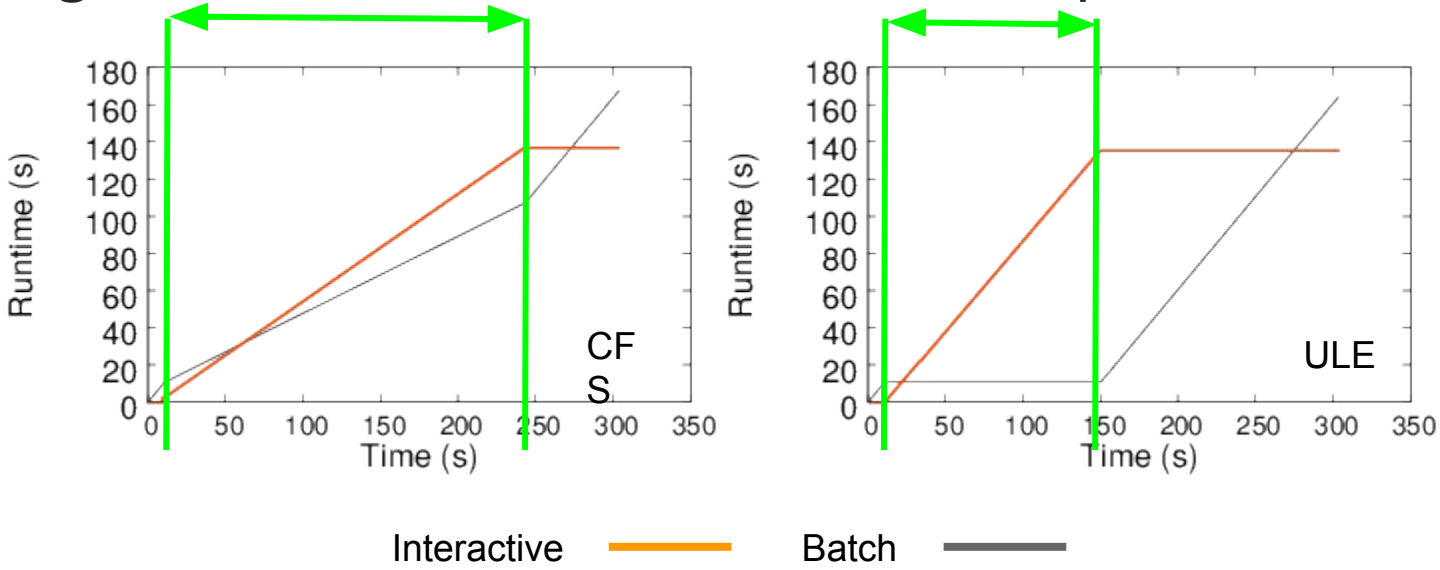
- Linux CFS is fair (both application get ~50% of the CPU)
- No starvation

Dealing with interactive tasks : The impact



- On FreeBSD ULE interactive tasks saturate the CPU and starve batch tasks !

Dealing with interactive tasks : The impact



- On ULE interactive applications may perform better ...
- ... But may also starve other tasks in the system.

Dealing with interactive tasks : “Auto-Starvation”

- Starvation problem in ULE can occur between threads of a single application
- Can be good for performances as it avoids over-subscription of the CPU!
- More details in the paper

Dealing with interactive tasks : Summary

In ULE :

- Interactive tasks have absolute priority
- They also can starve batch tasks, even from the same application

In CFS :

- All tasks are treated the same
- Fairness
- No starvation

Difference #2/4 : Preemption

Full preemption : The difference

Should a waking up tasks preempt the running task ?

- Linux CFS : Full preemption is enabled, so yes, sometimes.
- FreeBSD ULE : No full preemption by default. Only kernel threads can preempt others.

What impact ?

Full preemption : The experiment

- Run a communication intensive workload (Apache) on a single core
- The workload consist of a load injector and workers that handle requests
- Compare the performances and look at low-level events with perf

Full preemption : The impact ?

- Apache workload performs better on ULE on single core.
- On CFS the request injector is preempted by the workers at every request
- Thus further requests are delayed, performances go down !

	Linux CFS	FreeBSD ULE
Preemption of injector by userland thread	> 2M	0
Total time (seconds)	257	185
Requests / second	3891	5405

Full preemption : Summary

ULE :

- No full preemption by default

CFS :

- Full preemption is enabled by default
- Can worsen performances in some surprising ways

Difference #3/4 :
Load balancing

Load balancer : The difference

A scheduler must balance the load on all cores

- Both schedulers have their own load balancing algorithm ...
- ... which differ in three main points

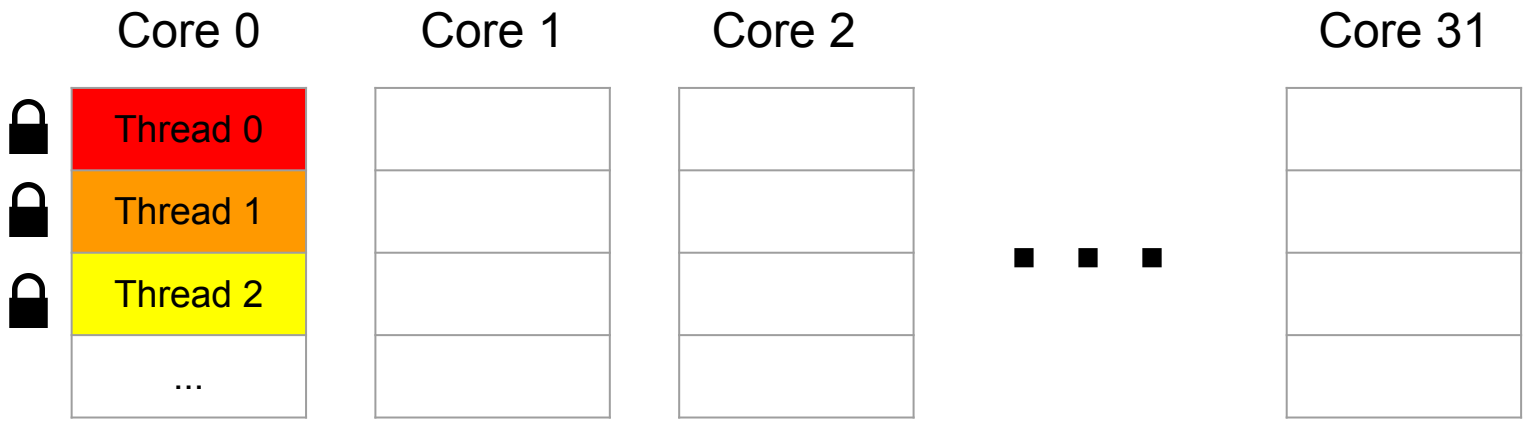
Linux CFS	FreeBSD ULE
Load = complex metric w/ heuristics	Load = number of tasks
Hierarchical (NUMA)	Non-hierarchical (SMP)
Every 4ms	Every 0.5-1.5s (random)
Migrates multiple tasks from a loaded core at once	Migrate at **most one task** from a loaded core

Load balancer : The experiment

- Use a lot of threads to put a lot of stress on the load balancer
- What we want to compare : The speed and the efficiency

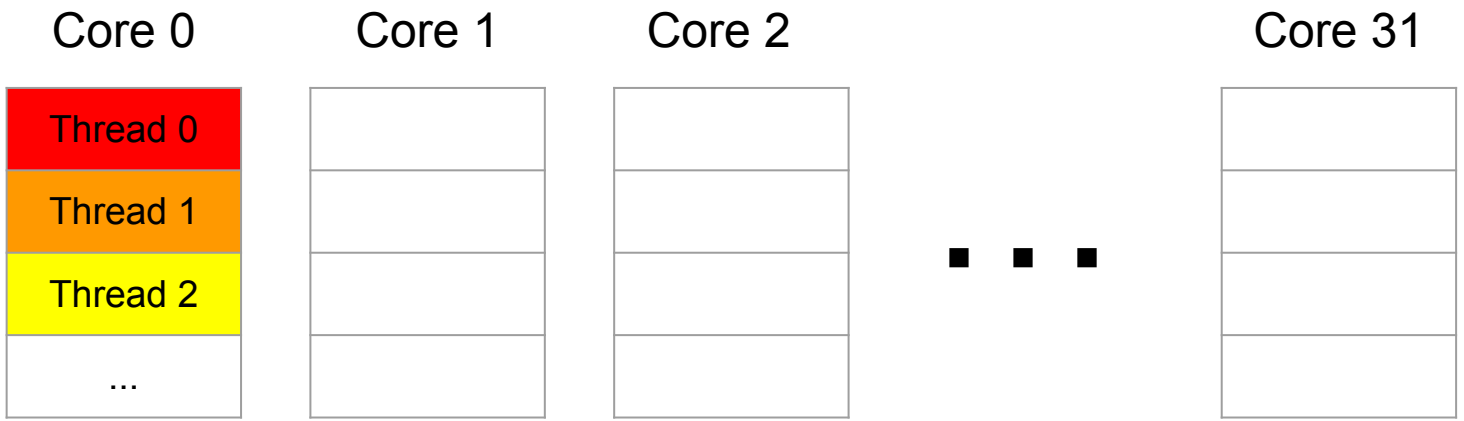
Load balancer : The experiment

1) Spawn a lot of threads, all pinned on core 0 of a 32-cores machine



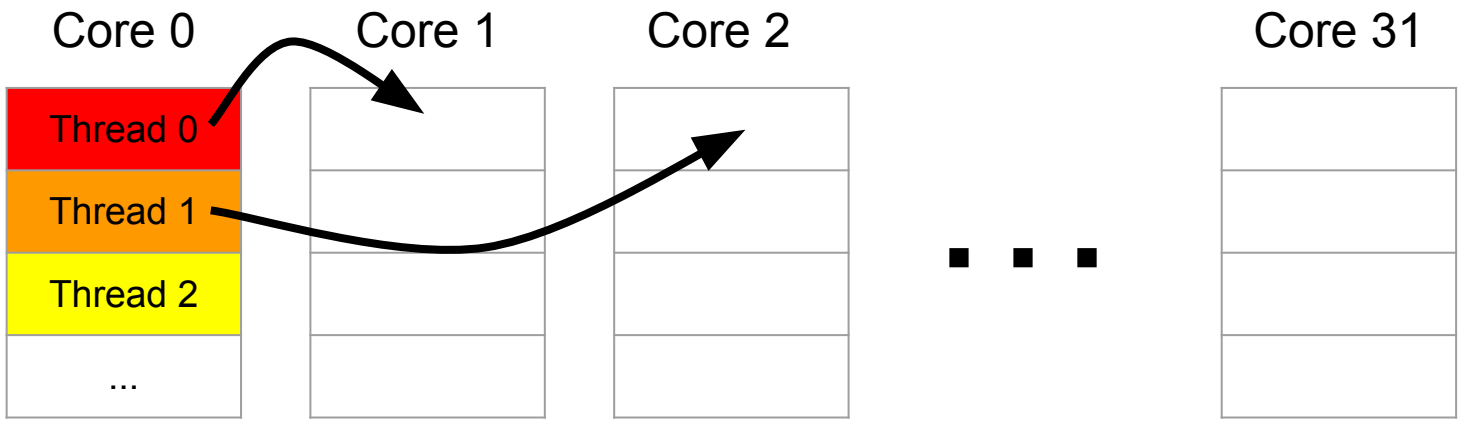
Load balancer : The experiment

2) Unpin the threads



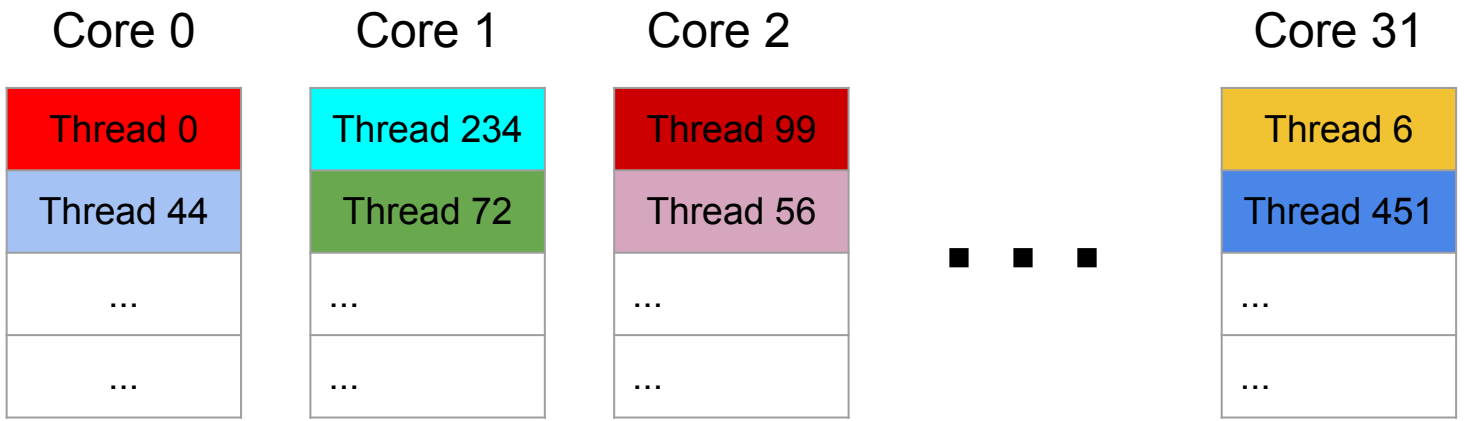
Load balancer : The experiment

3) Let the load balancer do its work and save runqueue sizes at every migration



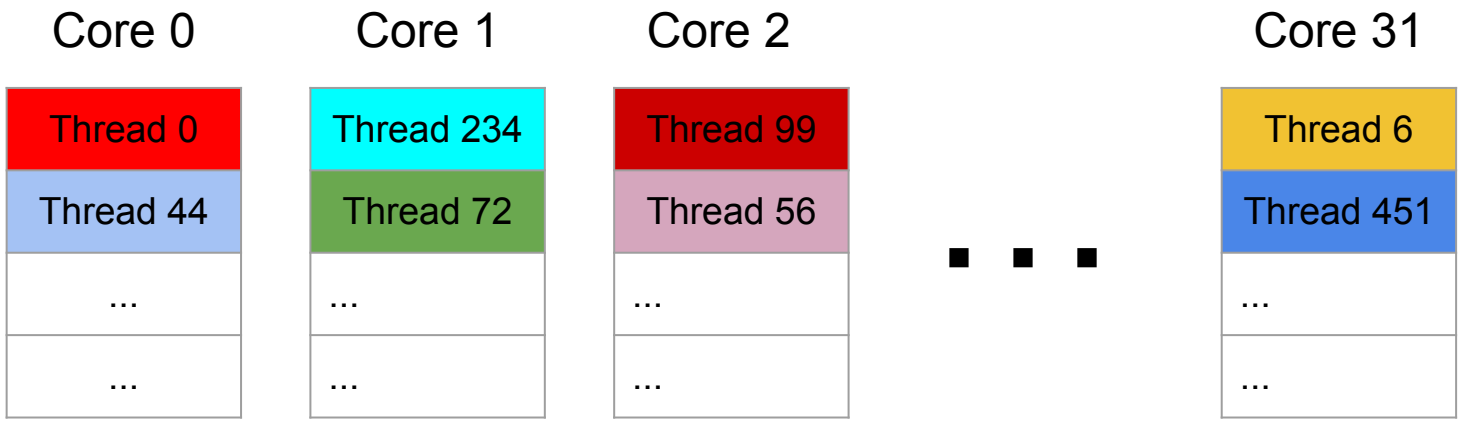
Load balancer : The experiment

4) ... until some stable state is reached



Load balancer : The experiment

5) Record the time it took to reach stable state

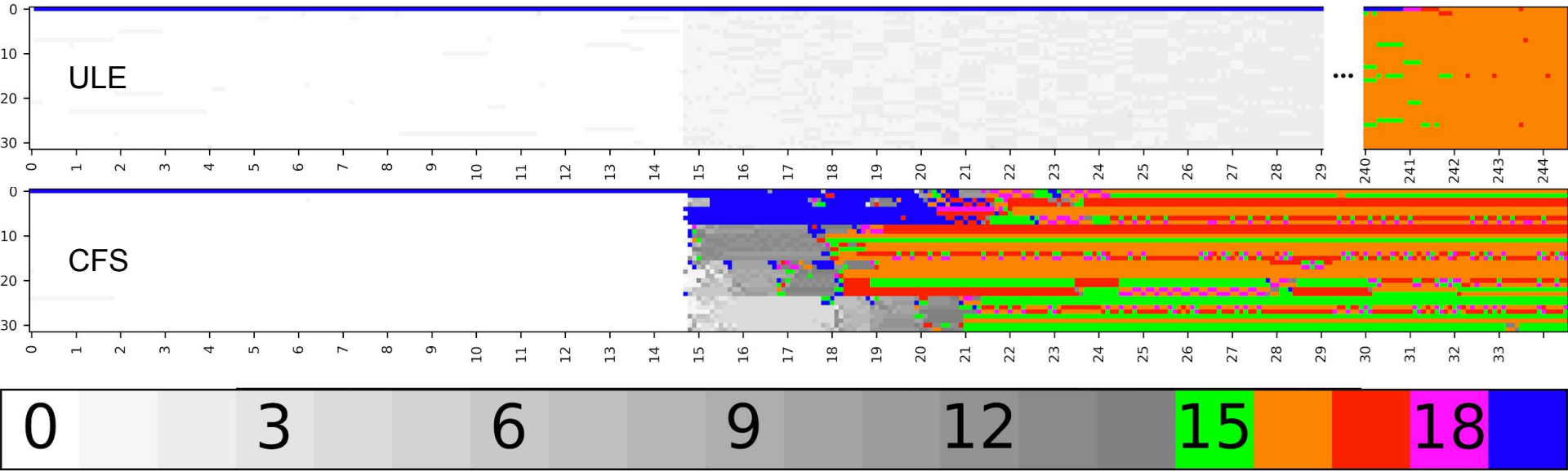


Load balancer : The experiment

- We used C-ray, a massively parallel ray-tracer with 512 threads that are all identical
- As all threads are identical, so we should expect 16 threads per core at the end...

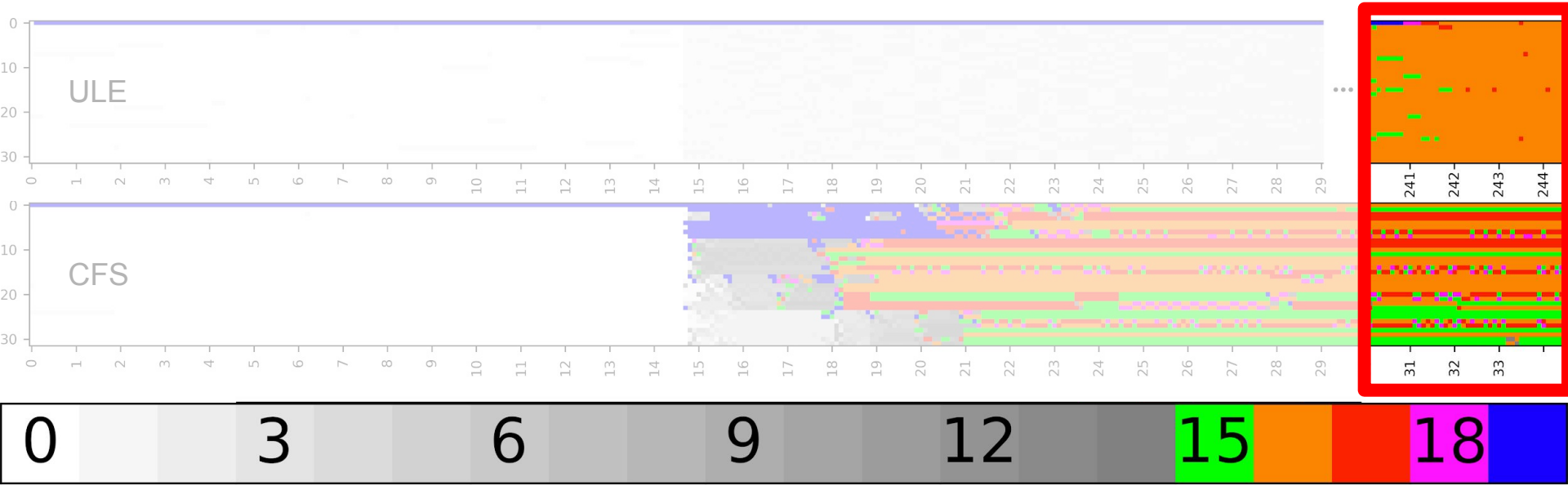
Load balancer : The impact

- We end up with the following graphs
- Each line is a core, the color is the size of its runqueue

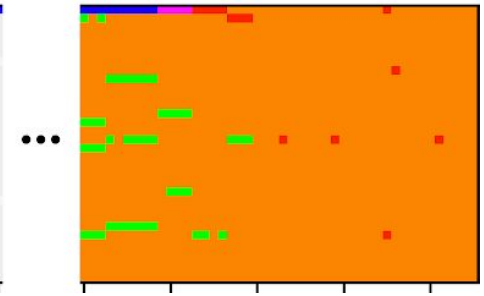


Load balancer : The impact

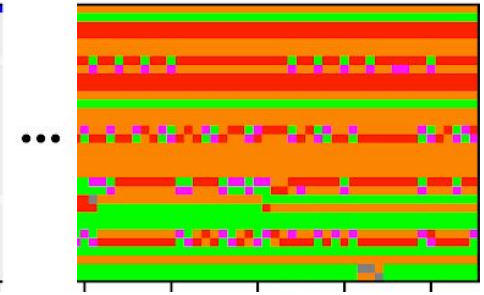
- We end up with the following graph
- Each line is a core, the color is the size of its runqueue



Load balancer : The impact, Perfect balancing



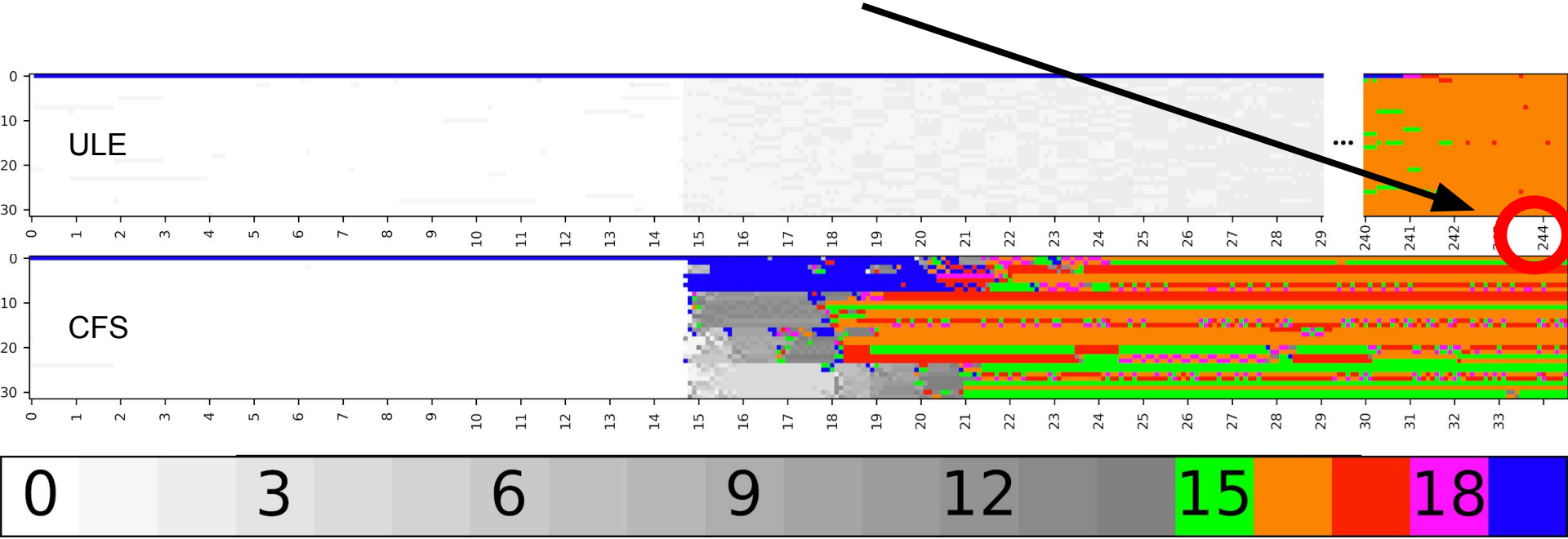
ULE achieves perfect balancing !



CFS has some troubles due to NUMA heuristics

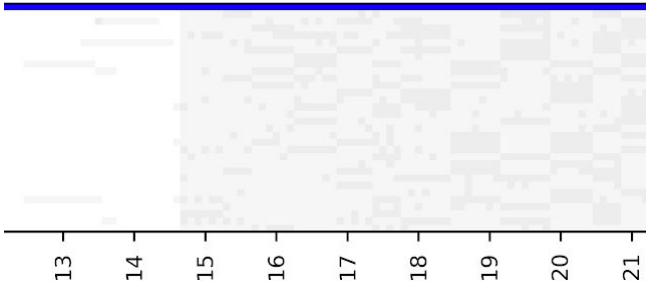
Load balancer : The impact, Speed

- Four minutes to spread the load on ULE ???



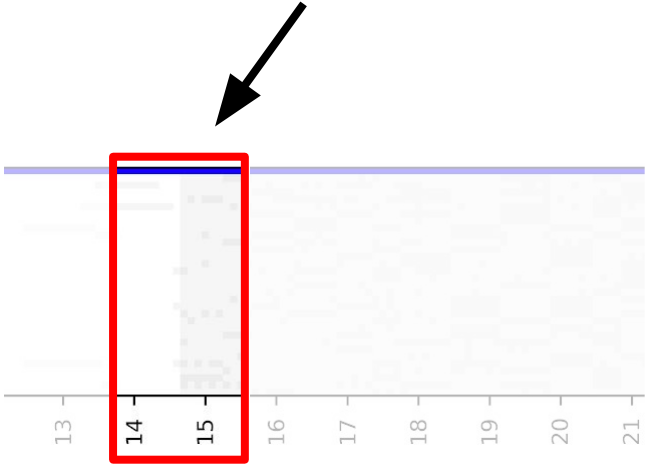
Load balancer : The impact, Speed (ULE)

- ULE can migrate at most one task from a loaded core !



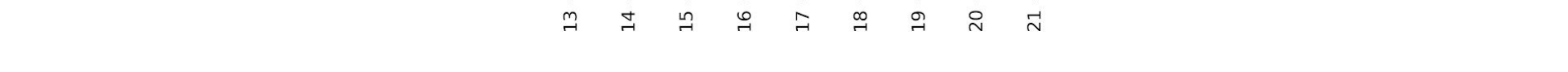
Load balancer : The impact, Speed (ULE)

- ULE can migrate at most one task from a loaded core !
- Idle cores also steal only one task at a time



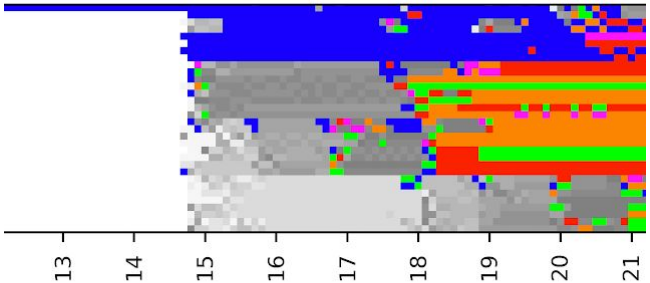
Load balancer : The impact, Speed (ULE)

- ULE can migrate at most one task from a loaded core !
- Idle cores also steal only one task at a time
- After the stealing, threads will be migrated one at a time ...



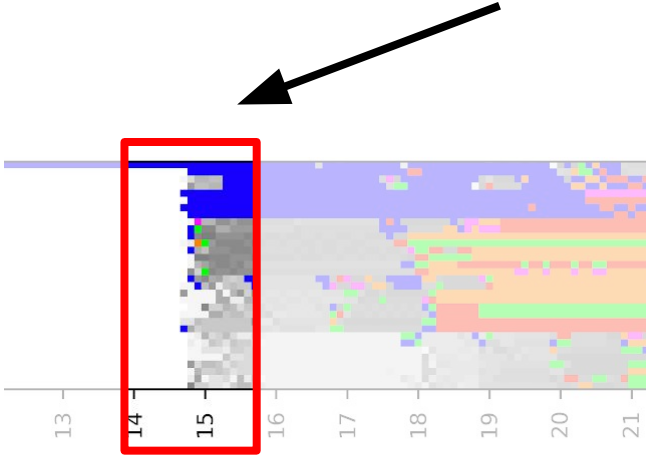
Load balancer : The impact, Speed (CFS)

- CFS has no limit on the number of migrations



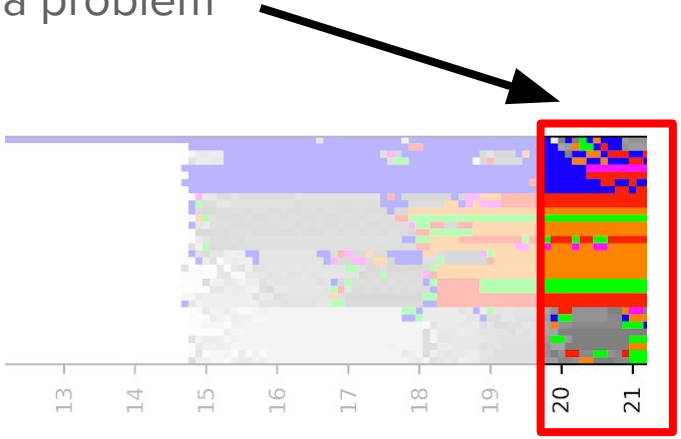
Load balancer : The impact, Speed (CFS)

- CFS has no limit on the number of migrations
- CFS balances the load much faster : around 400 migrations in less than 0.2s !



Load balancer : The impact, Speed (CFS)

- CFS has no limit on the number of migrations
- CFS balances the load much faster : around 400 migrations in less than 0.2s !
- But heuristics are still a problem



Load balancer : Summary

ULE :

- Very simple load metric
- Achieves perfect balancing
- But **slow**

CFS :

- Complex load metric, lots of heuristics
- Can be stuck in imbalanced state
- But **fast** at spreading the load

Difference #4/4 : Thread placement

Thread placement : The difference

How to choose the core running a new/waking thread ?

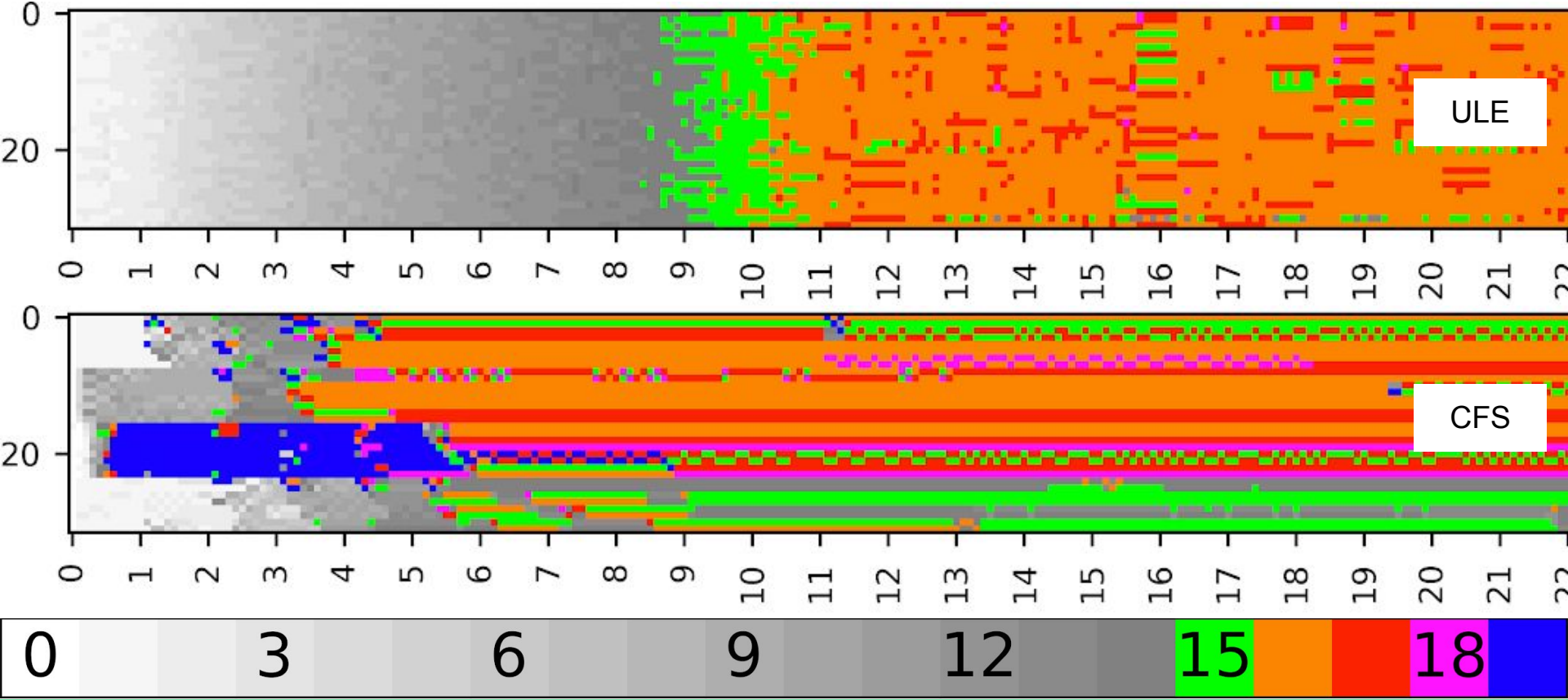
- CFS : Heuristic to restrict the list of suitable cores and take the less loaded one
- ULE : Choose, among **all** cores, the one with the minimum number of tasks

Thread placement : The experiment

- Spawn a lot of threads on all available cores
- Record size of runqueues over time

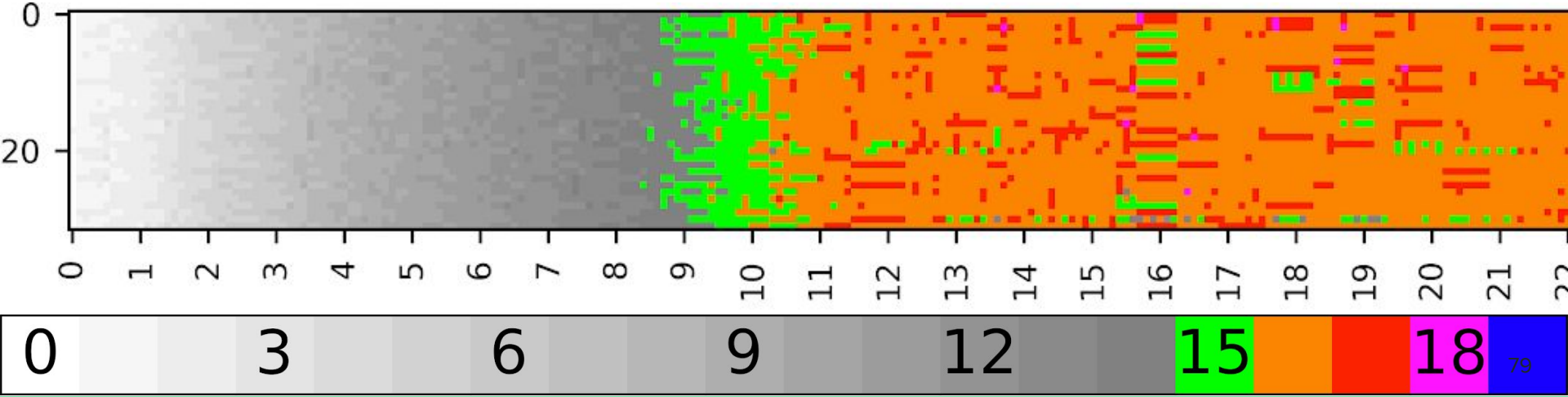
Again C-ray was a good choice for this

Thread placement : The impact



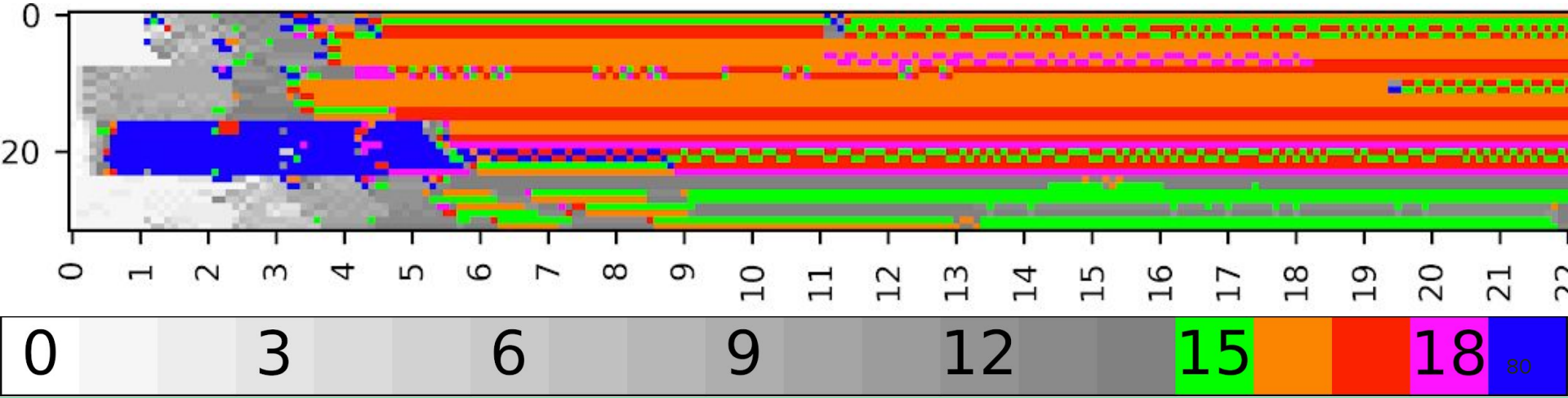
Thread placement : The impact : ULE

- The load is always more or less uniform (nice fading on the load graph)
- Auto-starvation slows down the creation of threads !



Thread placement : The impact : CFS

- Bad load balance at the beginning due to NUMA heuristics
- Load balancer tries to fix this but still struggles as before



Thread placement : Summary

ULE :

- Thread placement policy consider **all** cores ...
- ... and thus relieves the pressure from the load balancer
- Load is always uniform

CFS :

- The policy consider a **subset** of cores only using heuristics
- Might worsen the balancing in case of large spawn rates

Conclusion

Conclusion

- Scheduling is hard … and even harder on a multicore machines
- Design and implementation choices can have a great influence on performances
- No scheduler perform better than the other on all workloads

Questions ?

Code available on Github : https://github.com/JBouron/linux/tree/loadbalancing