https://arxiv.org/abs/2508.15703

A Linux CPU scheduler for interactive multi-threaded workloads

Al-Amjad Tawfiq Isstaif, Evangelia Kalyviannaki, Richard Mortier

Smart Sensing Lab @ Nottingham Trent University, Systems Research Group @ University of Cambridge

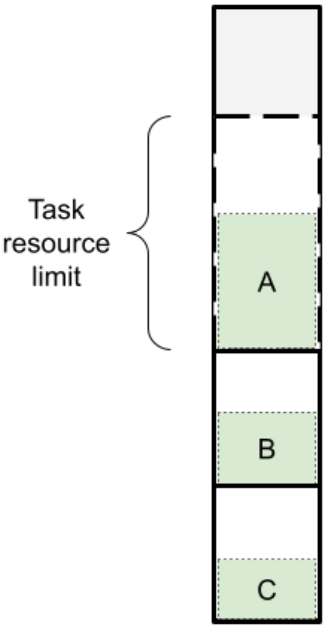
alamjad.isstaif@cl.cam.ac.uk

UNIVERSITY OF CAMBRIDGE

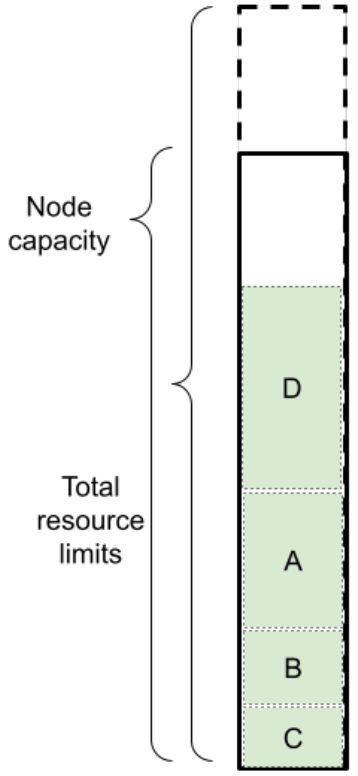
https://arxiv.org/abs/2508.15703

Motivation

- Originally interested in densely packing unikernels onto servers, aiming to improve on Kubernetes and serverless performance
- Measurements of serverless workloads made no sense: CPU utilisation was too high, throughput was too low – **performance was degraded**
- **Why?** With increasing workload density (10s or 100s of containers), **context-switch overhead takes up to 25% CPU time** due to how the scheduler manages cgroups
- **How to fix it?** We developed an alternative scheduler that mitigates this problem, allowing the same performance on a **20% smaller cluster**



a) Container packing based on CPU resource reservation



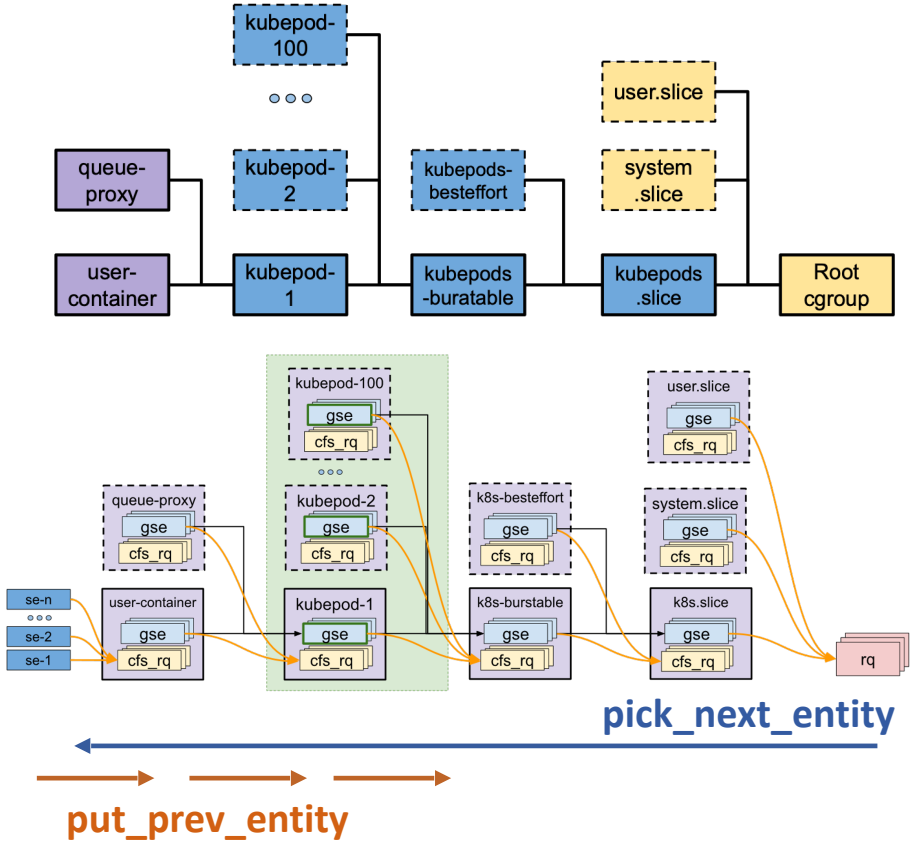
b) Container packing based on CPU multiplexing

UNIVERSITY OF CAMBRIDGE

https://arxiv.org/abs/2508.15703

Group scheduling with cgroups

- Group scheduling is the foundation of the **fair scheduler class** (i.e. kernel/sched/fair.c) which has been known as CFS and rebranded in recent kernels as EEVDF (v6.8+).
- The conservative baseline is that you pin a cgroup per each cpu core which can lead to wasted resources unlike with work-conserving scheduling
- Tasks are **grouped** by cgroup and scheduled as a whole using **per-cgroup** and **per-core** run queue structures to prevent gaming the scheduler.
- Locating the next task is optimised, but **re-inserting preempted tasks increases cost** of pick_next_task_fair by several microseconds
- Higher per-context-switch cost and rate of context-switches **combine multiplicatively**

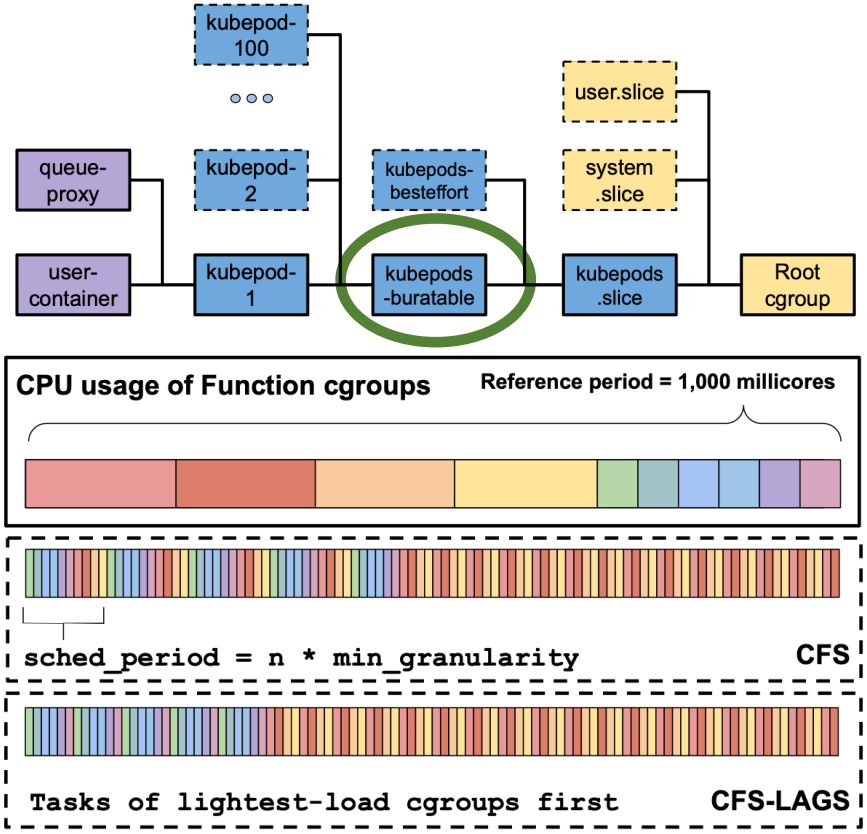


UNIVERSITY OF CAMBRIDGE

https://arxiv.org/abs/2508.15703

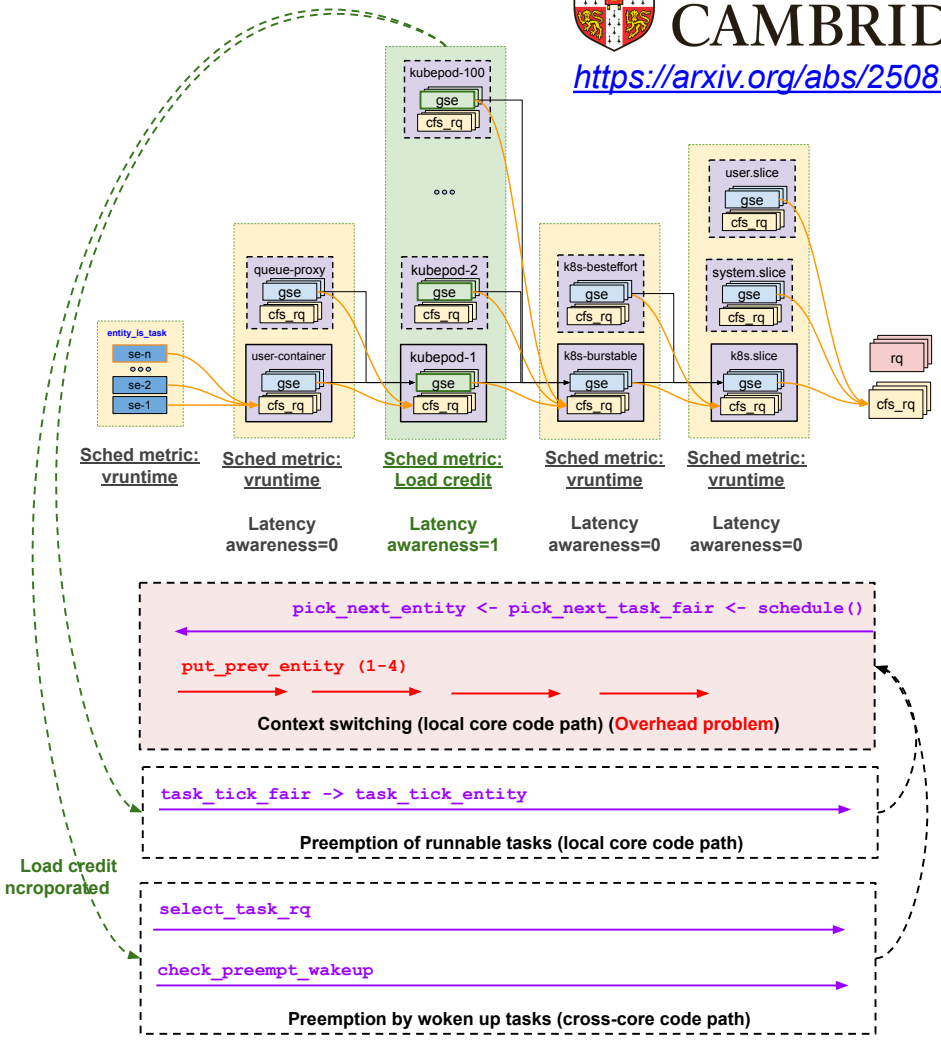
Latency-Aware Group Scheduling

- The key idea behind LAGS is to maximise **task completion rate within cgroups** beyond the scheduler's short scheduling interval
  - Use a Shortest-Remaining Time First approximation to **prioritise the lightest cgroups**
  - This improves latency and reduces overhead as **cgroups can exit the system earlier making runqueues shorter**
- Approximation via **Cgroup Load Credit** metric that tracks recent CPU usage for all threads within a cgroup



Realising CFS–LAGS

- CFS-LAGS is a **sub-scheduling architecture** applying custom policies to specific
  - **Load Credit metric** becomes the scheduling priority for serverless function cgroups
  - The default CFS/EEVDF policy remains for all other cgroups (i.e. virtual runtime or virtual deadline).
- **Load Credit implementation** averages the default *per-entity load tracking* (PELT) over a ~4s window *vs* default 32ms in PELT, preferring youngest tasks firs
- **User space interface** identifies target group tasks (that is, cgroups) via a user space `cpu.latency_awareness` cgroup property



UNIVERSITY OF CAMBRIDGE

https://arxiv.org/abs/2508.15703

E2E cluster performance (single node)

- **Higher throughput with lower latency**: LAGS increases throughput by 26% over CFS and 12% over EEVDF, while reducing median and tail latency by ~6x.
- **Doing more with less CPU cycles**. CPU utilisation drops from ~75-90% to ~40-50% under LAGS, as less time is spent in context switching (i.e. schedule()).

Scheduler	50th (ms)	95th (ms)	RPS	CPU (%)
CFS	590	1400	37.6	~90%
CFS-LLF	54	100	47.5	~40%
EEVDF	240	750	42.8	~75%
EEVDF-LLF	60	420	46.4	~50%

Table 1. Comparison of CFS and EEVDF with and without the LAGS approach under the Locust benchmark (§5.2)

aati2@caelum-301: ~ (ssh)

Name	# reqs	# fails	Avg	Min	Max	Median	req/s	failures/s
POST /predict	5086	1(0.02%)	327	12	2322	77	38.00	0.00
Aggregated	5086	1(0.02%)	327	12	2322	77	38.00	0.00
POST /predict	5189	1(0.02%)	323	12	2322	76	39.30	0.00
Aggregated	5189	1(0.02%)	323	12	2322	76	39.30	0.00
POST /predict	5252	1(0.02%)	322	12	2322	76	40.90	0.00
Aggregated	5252	1(0.02%)	322	12	2322	76	40.90	0.00
POST /predict	5339	1(0.02%)	329	12	2322	78	41.70	0.00
Aggregated	5339	1(0.02%)	329	12	2322	78	41.70	0.00
POST /predict	5427	1(0.02%)	326	12	2322	77	40.20	0.00
Aggregated	5427	1(0.02%)	326	12	2322	77	40.20	0.00
POST /predict	5488	1(0.02%)	330	12	2322	79	43.10	0.00
Aggregated	5488	1(0.02%)	330	12	2322	79	43.10	0.00
POST /predict	5577	1(0.02%)	337	12	2322	81	40.80	0.00
Aggregated	5577	1(0.02%)	337	12	2322	81	40.80	0.00
POST /predict	5632	1(0.02%)	339	12	2322	83	39.50	0.00
Aggregated	5632	1(0.02%)	339	12	2322	83	39.50	0.00
POST /predict	5716	1(0.02%)	349	12	2365	86	38.10	0.00
Aggregated	5716	1(0.02%)	349	12	2365	86	38.10	0.00
POST /predict	5789	1(0.02%)	352	12	2365	90	38.00	0.00
Aggregated	5789	1(0.02%)	352	12	2365	90	38.00	0.00

aati2@caelum-408: ~ (ssh)

```
  0[||||||||||||||100.0%]   3[||||||||||||||100.0%]   6[||||||||||||||100.0%]   9[||||||||||||||100.0%]
  1[||||||||||||||100.0%]   4[||||||||||||||100.0%]   7[||||||||||||||100.0%]  10[||||||||||||||100.0%]
  2[||||||||||||||100.0%]   5[||||||||||||||100.0%]   8[||||||||||||||100.0%]  11[||||||||||||||100.0%]
Mem[|||||||||||||||||||||                25.4G/62.8G] Tasks: 748, 9412 thr, 187 kthr; 12 running
Swp[                                       0K/0K] Load average: 31.39 18.48 11.48
                                                  Uptime: 90 days, 16:21:04
```

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
285161	aati2	20	0	20644	17040	4260	R	32.3	0.0	8:48.87	htop
4074274	1034	20	0	3002M	200M	101M	S	14.2	0.3	0:54.30	/opt/conda/bin/python /opt/conda/bin/bentoml
4068619	1034	20	0	3067M	200M	101M	R	10.6	0.3	0:49.30	/opt/conda/bin/python /opt/conda/bin/bentoml
4067884	1034	20	0	3002M	200M	101M	S	8.4	0.3	0:57.81	/opt/conda/bin/python /opt/conda/bin/bentoml
4070928	1034	20	0	2937M	199M	101M	S	8.4	0.3	0:48.68	/opt/conda/bin/python /opt/conda/bin/bentoml
285851	1034	20	0	3002M	200M	101M	S	8.0	0.3	0:04.78	/opt/conda/bin/python /opt/conda/bin/bentoml
4064677	1034	20	0	3002M	197M	99.1M	S	8.0	0.3	0:49.50	/opt/conda/bin/python /opt/conda/bin/bentoml
4064678	1034	20	0	3002M	199M	100M	S	8.0	0.3	0:50.12	/opt/conda/bin/python /opt/conda/bin/bentoml
4072825	1034	20	0	3068M	201M	101M	R	8.0	0.3	0:55.86	/opt/conda/bin/python /opt/conda/bin/bentoml
4074566	1034	20	0	3068M	201M	102M	S	8.0	0.3	0:54.26	/opt/conda/bin/python /opt/conda/bin/bentoml
4067393	1034	20	0	3067M	200M	101M	S	7.5	0.3	0:56.37	/opt/conda/bin/python /opt/conda/bin/bentoml
4070333	1034	20	0	2936M	199M	101M	S	7.5	0.3	0:49.37	/opt/conda/bin/python /opt/conda/bin/bentoml
4071763	1034	20	0	3067M	200M	101M	R	7.5	0.3	0:54.73	/opt/conda/bin/python /opt/conda/bin/bentoml
285852	1034	20	0	3002M	200M	101M	S	7.1	0.3	0:04.77	/opt/conda/bin/python /opt/conda/bin/bentoml
4067399	1034	20	0	3002M	200M	101M	S	7.1	0.3	0:52.00	/opt/conda/bin/python /opt/conda/bin/bentoml
4069265	1034	20	0	3002M	200M	101M	S	7.1	0.3	0:55.70	/opt/conda/bin/python /opt/conda/bin/bentoml
285848	1034	20	0	3002M	200M	101M	R	6.6	0.3	0:04.71	/opt/conda/bin/python /opt/conda/bin/bentoml
285849	1034	20	0	3002M	200M	101M	S	6.6	0.3	0:04.74	/opt/conda/bin/python /opt/conda/bin/bentoml
285850	1034	20	0	3002M	200M	101M	S	6.6	0.3	0:04.80	/opt/conda/bin/python /opt/conda/bin/bentoml
4064681	1034	20	0	2937M	198M	100M	S	6.6	0.3	0:49.53	/opt/conda/bin/python /opt/conda/bin/bentoml
4070702	1034	20	0	3002M	200M	101M	S	6.6	0.3	0:49.85	/opt/conda/bin/python /opt/conda/bin/bentoml
4073888	1034	20	0	3067M	200M	101M	S	6.6	0.3	0:56.09	/opt/conda/bin/python /opt/conda/bin/bentoml
286153	1034	20	0	3067M	200M	101M	R	6.2	0.3	0:04.81	/opt/conda/bin/python /opt/conda/bin/bentoml
4068147	1034	20	0	3002M	200M	101M	S	6.2	0.3	0:51.11	/opt/conda/bin/python /opt/conda/bin/bentoml
4070811	1034	20	0	3002M	200M	101M	S	6.2	0.3	0:48.42	/opt/conda/bin/python /opt/conda/bin/bentoml
285916	1034	20	0	3002M	200M	101M	S	5.8	0.3	0:05.04	/opt/conda/bin/python /opt/conda/bin/bentoml
286152	1034	20	0	3067M	200M	101M	R	5.8	0.3	0:04.75	/opt/conda/bin/python /opt/conda/bin/bentoml
4065038	1034	20	0	3002M	198M	99.9M	R	5.8	0.3	0:51.09	/opt/conda/bin/python /opt/conda/bin/bentoml
4071095	1034	20	0	2937M	199M	101M	R	5.8	0.3	0:50.66	/opt/conda/bin/python /opt/conda/bin/bentoml

F1Help F2Setup F3SearchF4FilterF5Tree F6SortByF7Nice -F8Nice +F9Kill F10Quit

aati2@caelum-408: /local/scratch/CFS-LLF_main (ssh)

```
/sys/fs/cgroup/*/*/*/*/cpu.weight
1024
1024
1024
1024
--------
resetting latency awawreness flags
/sys/fs/cgroup/*/cpu.latency_awareness
/sys/fs/cgroup/*/*/cpu.latency_awareness
/sys/fs/cgroup/*/*/*/cpu.latency_awareness
/sys/fs/cgroup/*/*/*/*/cpu.latency_awareness
0
0
0
0
-----
aati2@caelum-408:/local/scratch/CFS-LLF_main$
```

Demo available at
https://github.com/isstaif/CFS-LLF_main/blob/main/cluster/README.md

UNIVERSITY OF CAMBRIDGE

https://arxiv.org/abs/2508.15703

CFS-LAGS mitigates overhead

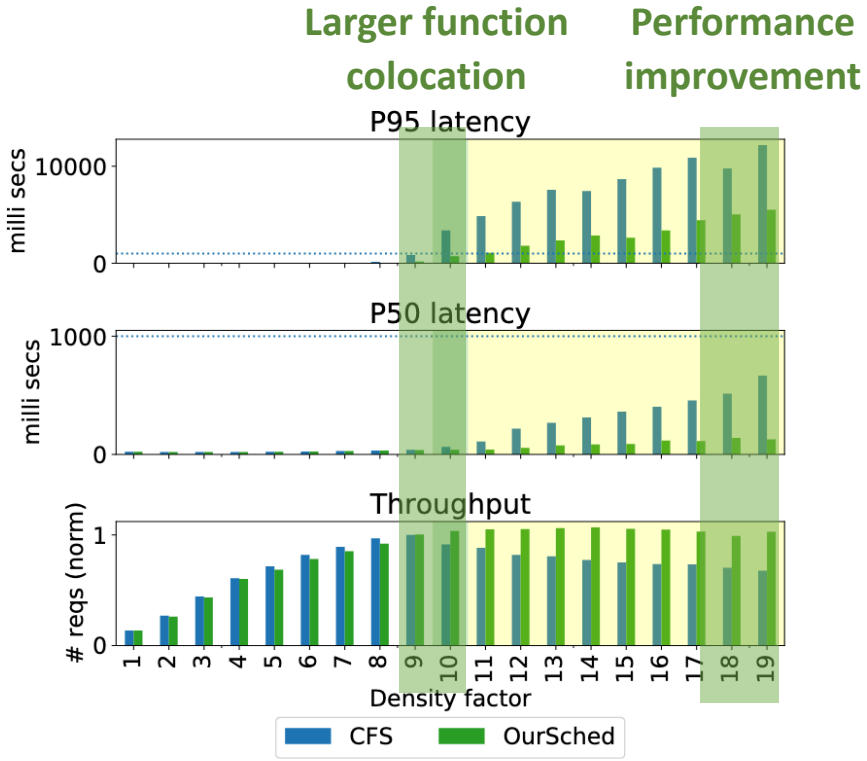


Figure 9. Performance given increasing colocated functions under realistic arrivals (azure2021).

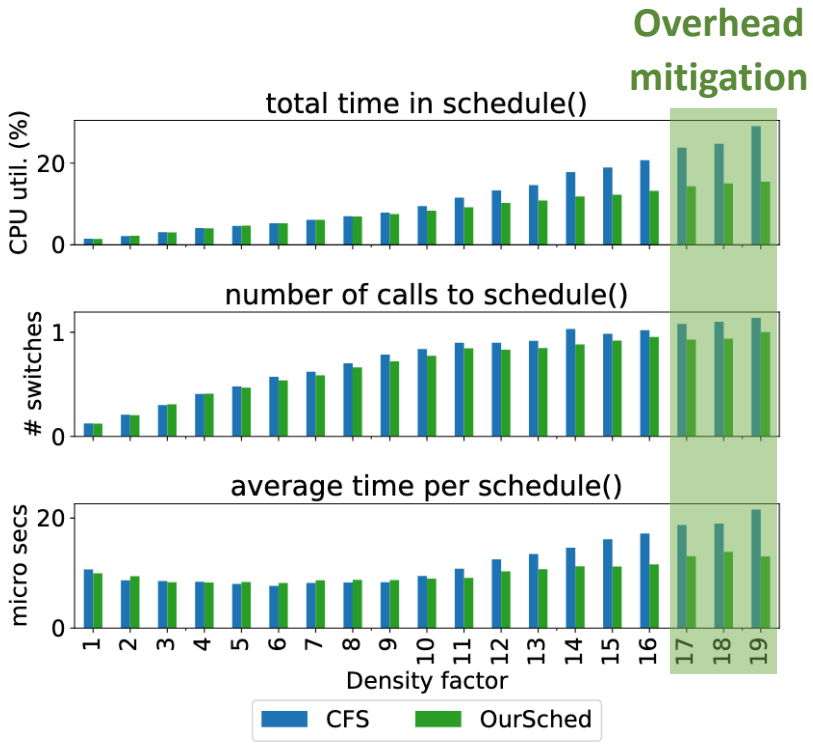


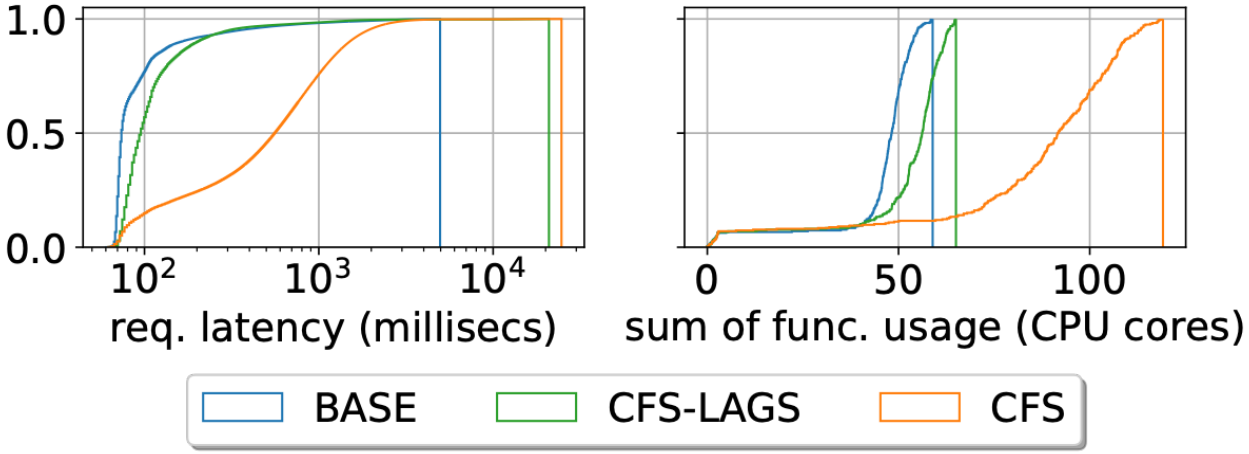
Figure 10. Scheduling overhead given increasing colocated functions under realistic arrivals (azure2021).

UNIVERSITY OF CAMBRIDGE

https://arxiv.org/abs/2508.15703

CFS-LAGS for tighter container packing

- **BASE** baseline of an provisioned cluster load. No CPU resource sharing, low CPU utilisation **(~30%)** given ~800 containers from **azure2021** (**~60 cores**)
- **CFS** used to statistically multiplex containers onto 12 nodes. Max density achievable given requirement to **keep CPU utilisation below ~45%**
- **CFS-LAGS** achieves the same performance with 10 nodes and **higher CPU utilisation (~55%)**



CFS-LAGS cluster (10 nodes)

Cluster-wide latency
Given the same density under CFS, the scheduling overheads translates into ~6x increase in latency!

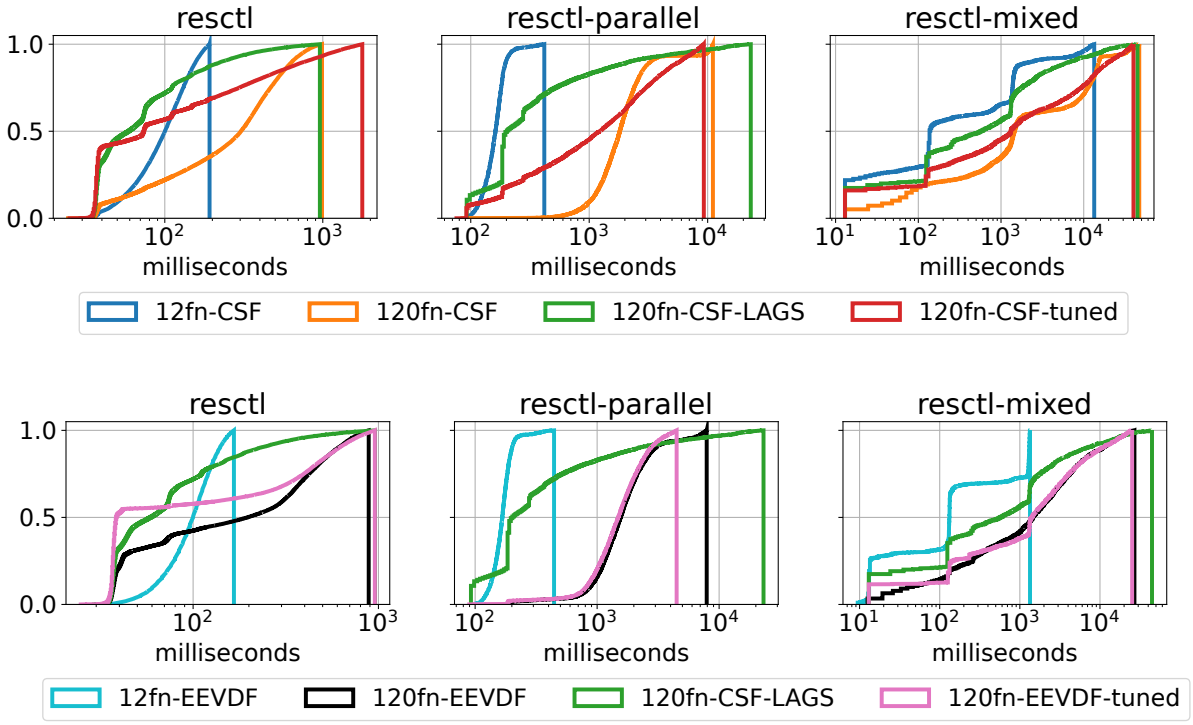
Cluster-wide CPU utilisation
CFS creates a significant gap between **effective** and **perceived** utilisation:
+100%, ~120 cores for CFS!
vs +10%, ~65 cores for CFS-LAGS

UNIVERSITY OF CAMBRIDGE

https://arxiv.org/abs/2508.15703

Comparing CFS-LAGS to alternatives

- Simply increasing timeslice to 100ms increases cgroup-level task completion and improves latency but not for multi-threaded workloads (`resctl-parallel` and `resctl-mixed`)
- EEVDF is difficult to tune under high load (`120fn-EEVDF` and `120fun-EEVDF-tuned`) due to the virtual deadline scheduling used to enforce thread-level fairness



UNIVERSITY OF CAMBRIDGE

https://arxiv.org/abs/2508.15703

Conclusions

- Cgroups are crucial to manage workloads in production but
  - ...as load increases, reinsertion into the nested red-black tree structure increases context-switch times which
  - ...which in turn increases the context-switch rate as CFS tries to achieve fairness
  - ...and these two effects combine multiplicatively
- **This increases latency variation dramatically while decreasing effective capacity leading to poor bin-packing decisions by the K8s scheduler**
  - Appears to be mitigated currently by expensive, wasteful over-provisioning
  - Even worse for serverless where desire to avoid cold-starts leads to artificially increasing multiplexing by keeping idle tasks around
- CFS-LAGS unlocks 10---20% capacity by building on CFS/EEVDF while **using cgroups as user-space control interface** to encourage short tasks to complete early, keeping runqueues shorter, reducing context switch overhead, and improving binpacking

UNIVERSITY OF CAMBRIDGE

https://arxiv.org/abs/2508.15703

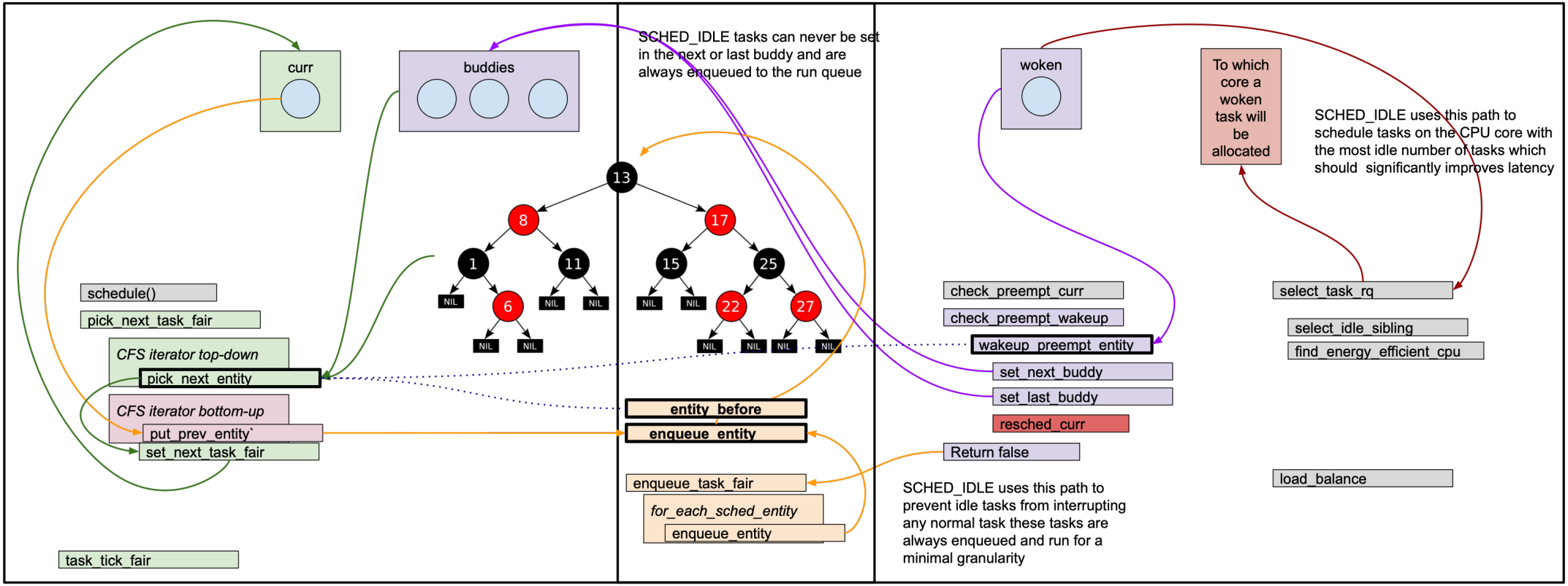
Status

- LAGS ported to kernel v6.12 and extending its use to manage the **QoS for interactive, cgroup-managed, multi-threaded workloads**.
  - Patch and benchmarks available via https://github.com/isstaif/CFS-LLF_main
  - Paper available via https://arxiv.org/pdf/2508.15703
- Several recent attempts to customise group scheduling
  - E.g., *sched_ext* cgroup sub-scheduling in v6.18 and ScyllaDB's user-space scheduler
  - Exploring how far we can get using *sched_ext* (on ARM...)

UNIVERSITY OF CAMBRIDGE

https://arxiv.org/abs/2508.15703

CFS code paths



UNIVERSITY OF CAMBRIDGE

https://arxiv.org/abs/2508.15703

CFS group scheduling data structures