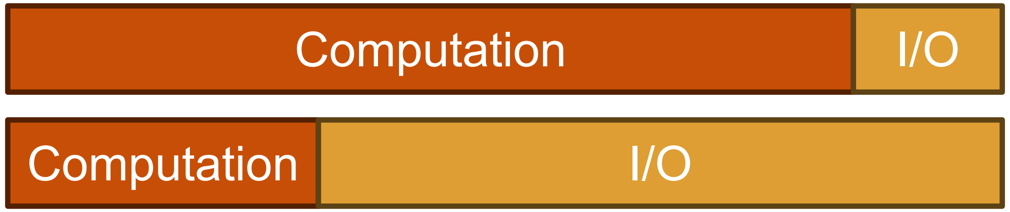
Tackling the Imbalance Between Computation and I/O

Taylan Özden, Hamid Fard, and Felix Wolf
Technical University of Darmstadt

Data-intensive applications

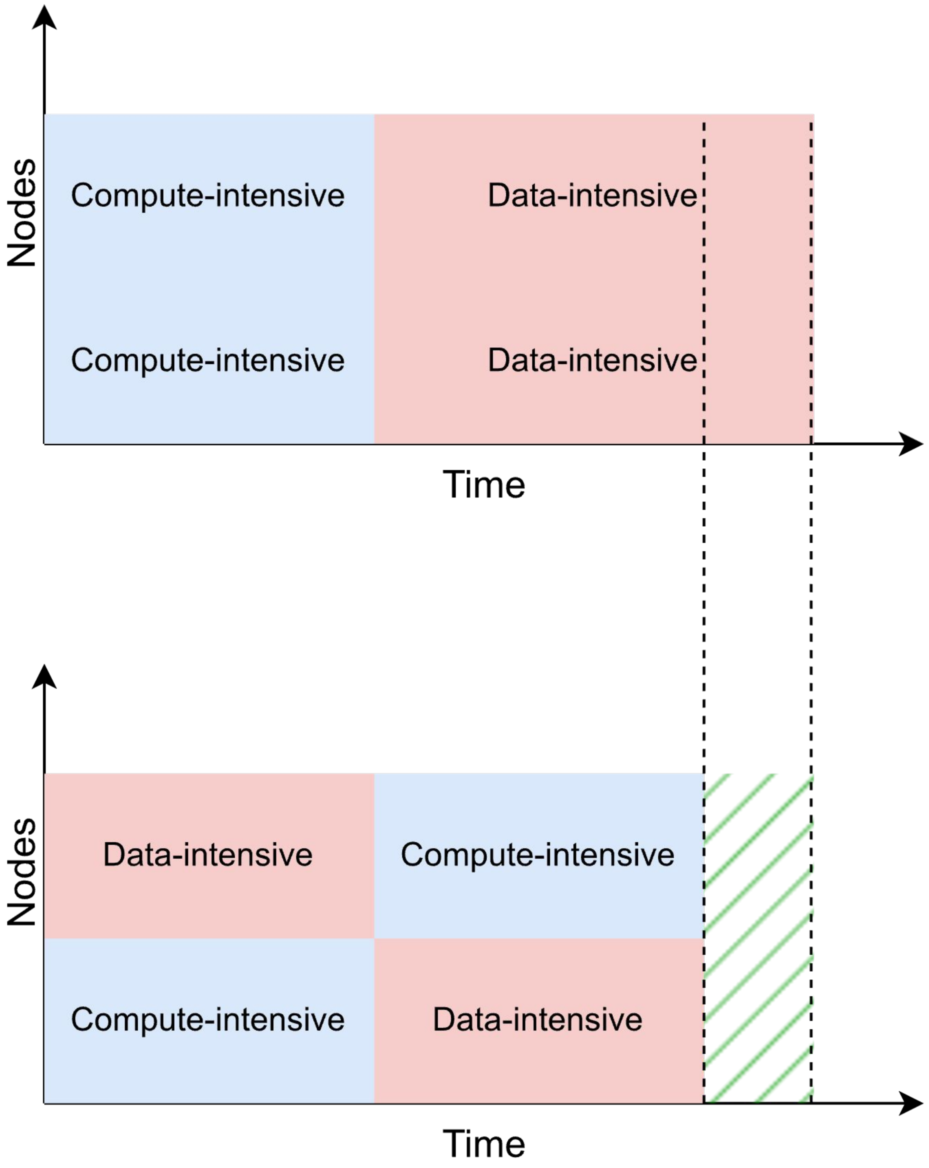


Compute bound

I/O bound

> **Amdahl's law**: ”The overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used."

- Consequence for **data-intensive** applications
  - They suffer more from low I/O bandwidth than compute-intensive ones

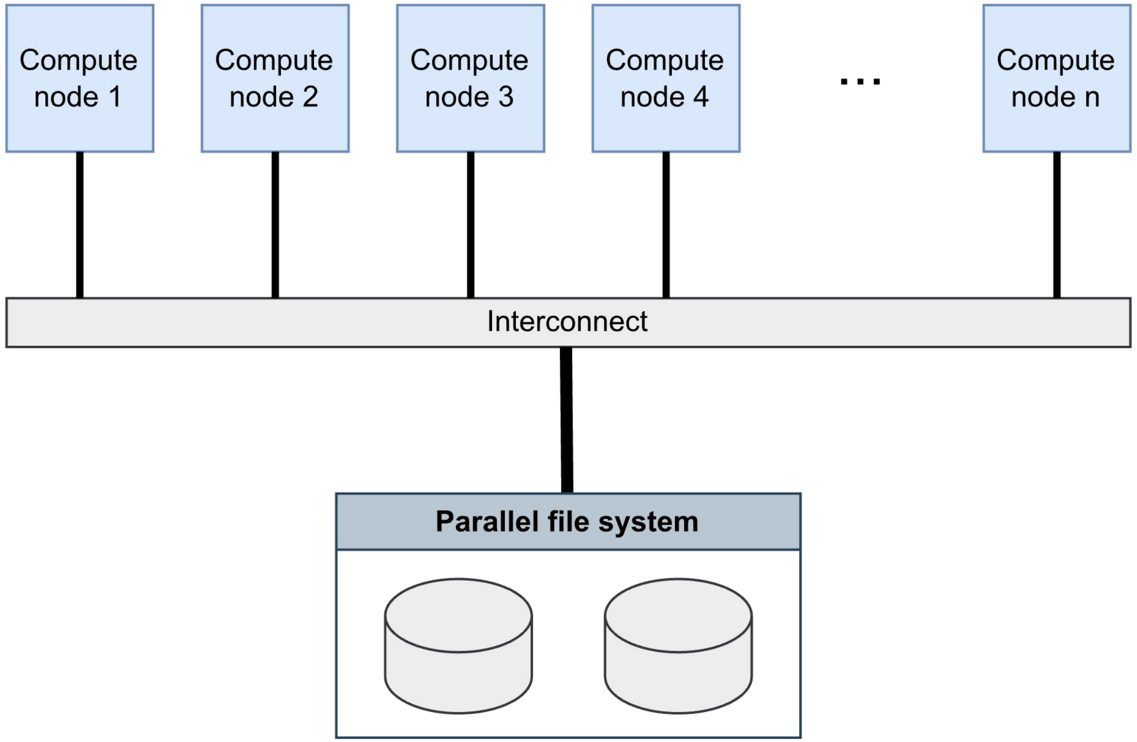


Imbalance between computation and I/O

- Compute-intensive applications can better tolerate data-intensive ones on their side
- Need a **scheduling algorithm** to avoid co-scheduling of data-intensive applications

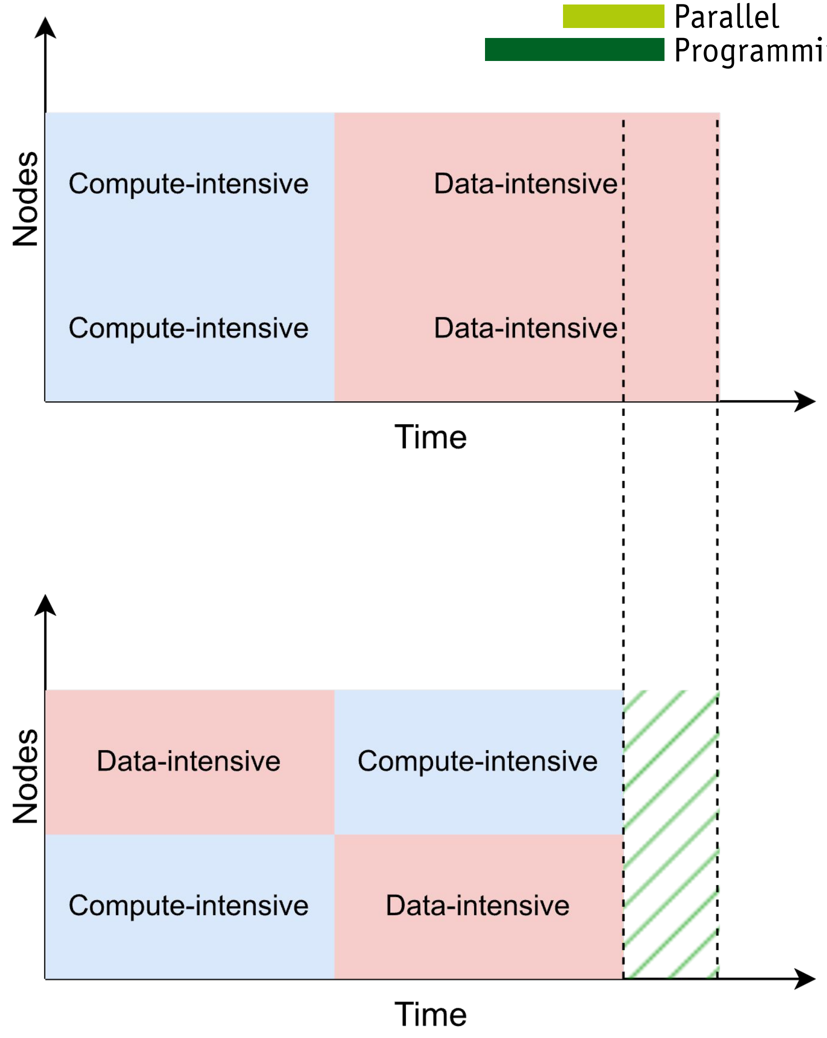
Assumptions

- No scheduling of I/O bandwidth
- **I/O intensity** of a job (roughly) known
- Applications have
  - exclusive access to compute nodes
  - shared access to the parallel file system (PFS)
- Combined scheduling of rigid and malleable jobs (no essential feature)



Proposed approach

- Overall goal - reduce makespan by keeping the I/O intensity of running jobs close to the average I/O intensity of the entire workload
- Specific objectives
  - Minimize PFS congestion
  - Ensure fairness to prevent job starvation
  - Exploit malleability



TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

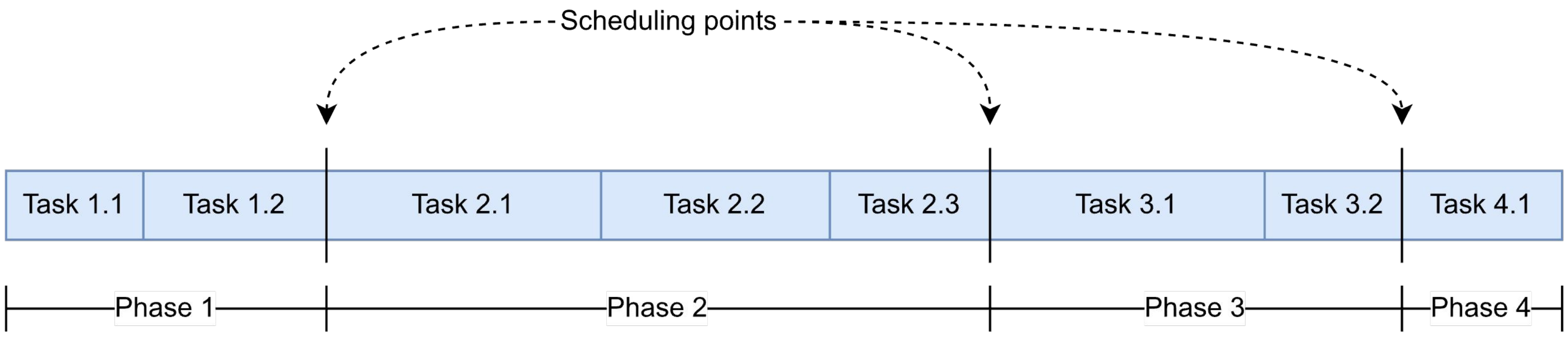
I/O intensity

- For all running jobs (R) and queued jobs (Q), we introduce three different I/O intensity measurements for
  - each job: $io\_intensity_j = \frac{io\_walltime_j}{total\_walltime_j} \cdot \emptyset bw_j$
  - the system: $io\_intensity(\mathbb{S}) = \frac{\sum_{j \in R}(intensity_j)}{|R|}$
  - the workload: $io\_intensity(\mathbb{W}) = \frac{\sum_{j \in R \cup Q}(intensity_j)}{|R \cup Q|}$

Under average conditions

Malleable jobs

- Malleable jobs are able to dynamically adapt to reconfiguration requests
- Reconfigurations occur at *scheduling points*, representing safe states where applications can shrink or expand resources



Scheduling algorithm

- Balances the I/O intensity of the executing workload by minimizing

$$|intensity(\mathbb{W}) - intensity(\mathbb{S})|$$

- The batch system invokes the scheduler at
  - job submission
  - job completion
  - scheduling points of malleable jobs
- Each event may modify the system or workload I/O intensity

Event	Affected I/O intensity metric
Job submission	
Job admission	
Job completion	
Job reconfiguration	

Preventing job starvation

- Decisions based purely on I/O intensity may cause starvation
- We introduce a weighted priority metric based on the I/O intensity of a job and its order of arrival
- Let $\alpha \epsilon [0,1]$ be the *reordering strength* the site administrator can choose with
  - $\alpha = 0$ representing first-come first-serve (FCFS)
  - $\alpha = 1$ maximum optimization for I/O intensity

Fairness priority

- As we consider malleability, we derive our proposed priority metric for the set of candidates $C = Q \cup R$
- For each candidate $c \in C$, we define pos_c, representing its relative position in the queue in the order of submission
- The scheduler calculates the fairness priority value $\lambda_c \in [0,1]$, starting with the first pending job in the queue

$$min\_pos = \min_{c \in C}(pos_c)$$

$$max\_pos = \max_{c \in C}(pos_c)$$

$$\lambda_c = \frac{pos_c - min\_pos}{max\_pos - min\_pos}$$

Weighted priority
(combines fairness with I/O intensity)

- For each candidate c, we calculate and normalize its intensity delta $\delta_c \in [0,1]$
- The scheduler calculates the weighted priority based on the reordering strength α
- In the final step, the scheduler chooses the best candidate, represented by the minimum weighted priority value, and schedules or reconfigures the job

$$io\_intensity_{new}(\mathbb{S}) = \frac{\sum_{j \in R}(io\_intensity_j) + io\_intensity_c}{|R| + 1}$$

$$\delta_c = |io\_intensity(\mathbb{W}) - io\_intensity_{new}(\mathbb{S})|$$

$$wp(\alpha, \lambda_c, \delta_c) \coloneqq (1 - \alpha) \cdot \lambda_c + \alpha \cdot normalized(\delta_c)$$

Algorithm

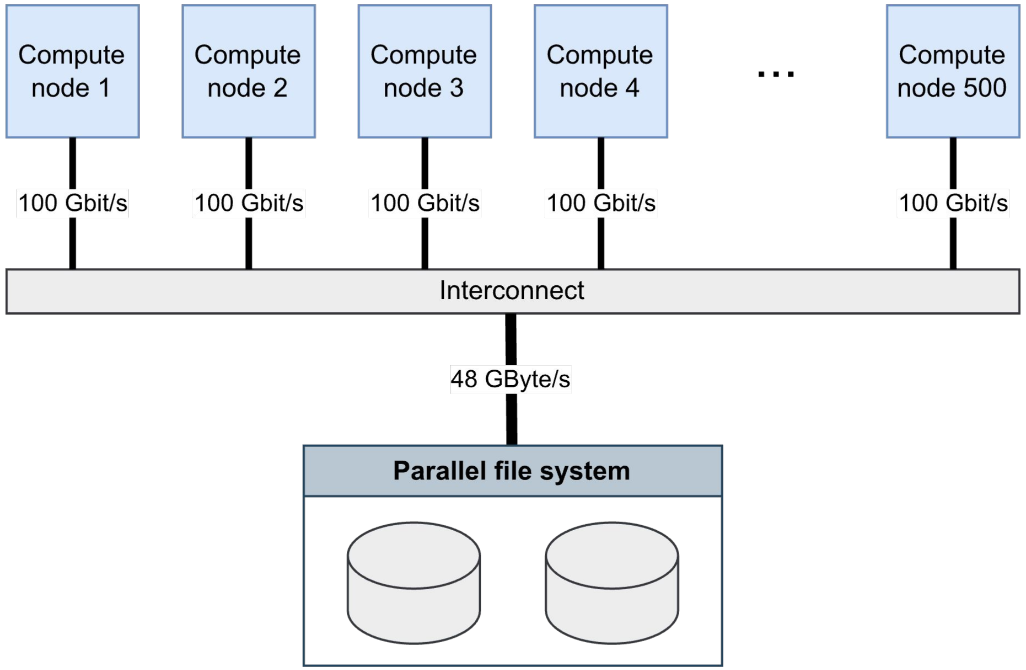
- At each invocation, we
  - Recalculate the I/O intensities
  - Make scheduling decisions based on the weighted priority
- For malleable applications, we calculate the I/O intensity for each configuration at each scheduling point
  - Expand malleable jobs if beneficial
  - Shrink only if suitable candidate in the queue

Input: List of jobs: J; list of nodes: N; invocation trigger: $trigger$; triggering job: j

```
if trigger ≠ SCHEDULING_POINT then
    if trigger = JOB_SUBMISSION then
        add_job_to_workload_io_intensity(j);
    else if trigger = JOB_COMPLETION then
        remove_job_from_system_io_intensity(j);
        remove_job_from_workload_io_intensity(j);
    end
    find_and_schedule_job(J, N);
else
    nodes_j^new ← get_best_configuration(j);
    if nodes_j^new ≠ {} then
        nodes_j^old ← nodes_j;
        nodes_j ← nodes_j^new;
        update_system_io_intensity(j, nodes_j^old);
        if nodes_j^new < nodes_j^old then
            find_and_schedule_job(J, N);
        end
    end
end
```

Validation via simulation

- We use ElastiSim, a batch-system simulator for malleable workloads
- 500 compute nodes, each with
  - Computing power of 100 GFLOP/s
  - Network link capacity of 100 Gbit/s
- Shared PFS with a peak bandwidth of 48 GB/s
- **Baseline**: malleable FCFS algorithm (FCFSm)



https://elastisim.github.io

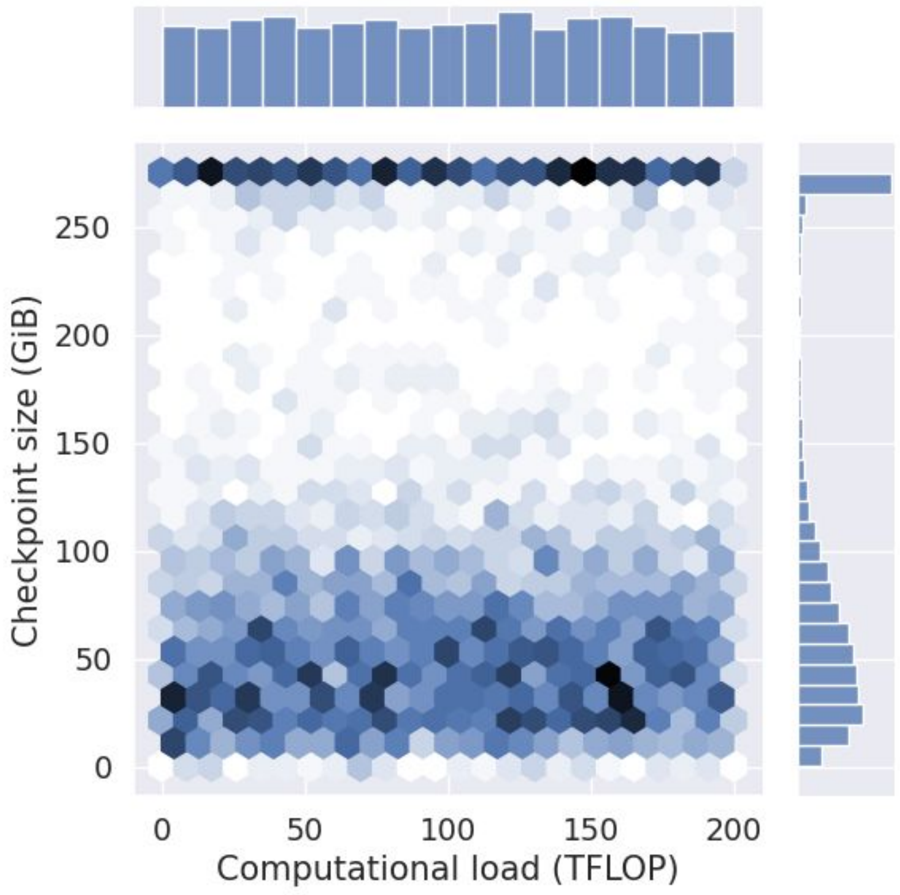
Taylan Özden, Tim Beringer, Arya Mazaheri, Hamid Mohammadi Fard, Felix Wolf: ElastiSim: A Batch-System Simulator for Malleable Workloads. In Proc. of the 51st International Conference on Parallel Processing (ICPP), Bordeaux, France, pages 1–11, ACM, August 2022 [DOI].

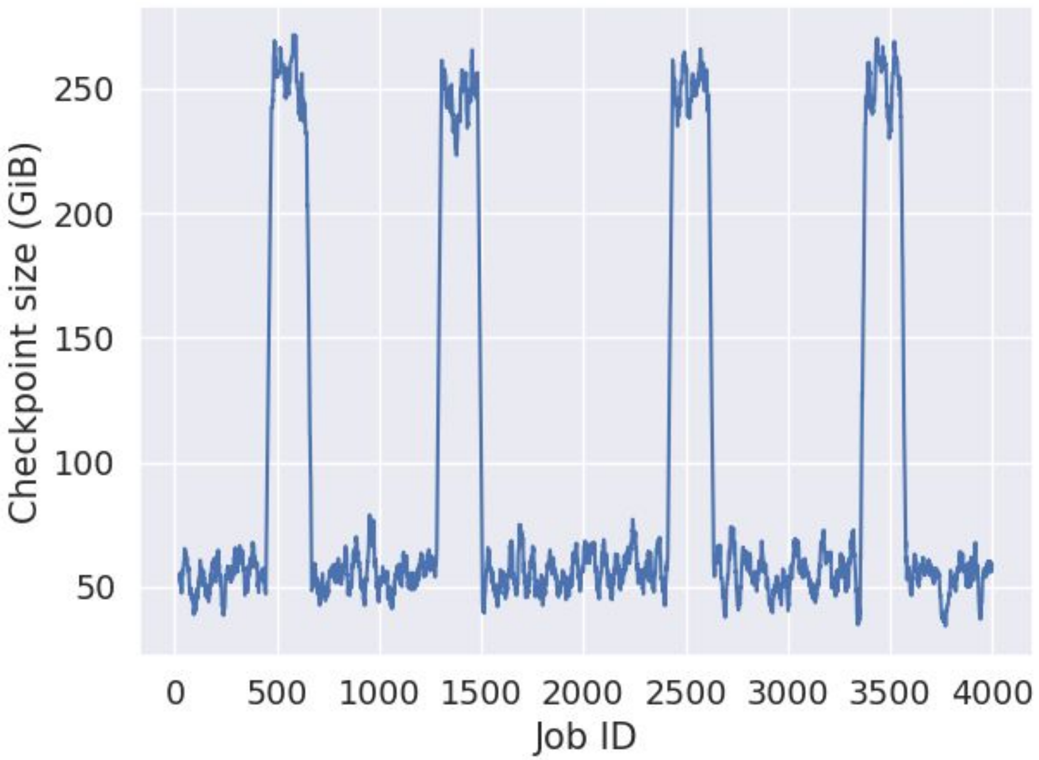
Simulated workload

- Our synthetic workload comprises 4000 jobs, with 80% rigid and 20% malleable jobs
- Each job repetitively runs a sequence of a compute and a checkpoint task (10–25 repetitions)
- Rigid jobs request a fixed number of nodes between 2 and 20, based on the job's computational and I/O load
- Malleable jobs support any configuration between 2 and 20 nodes

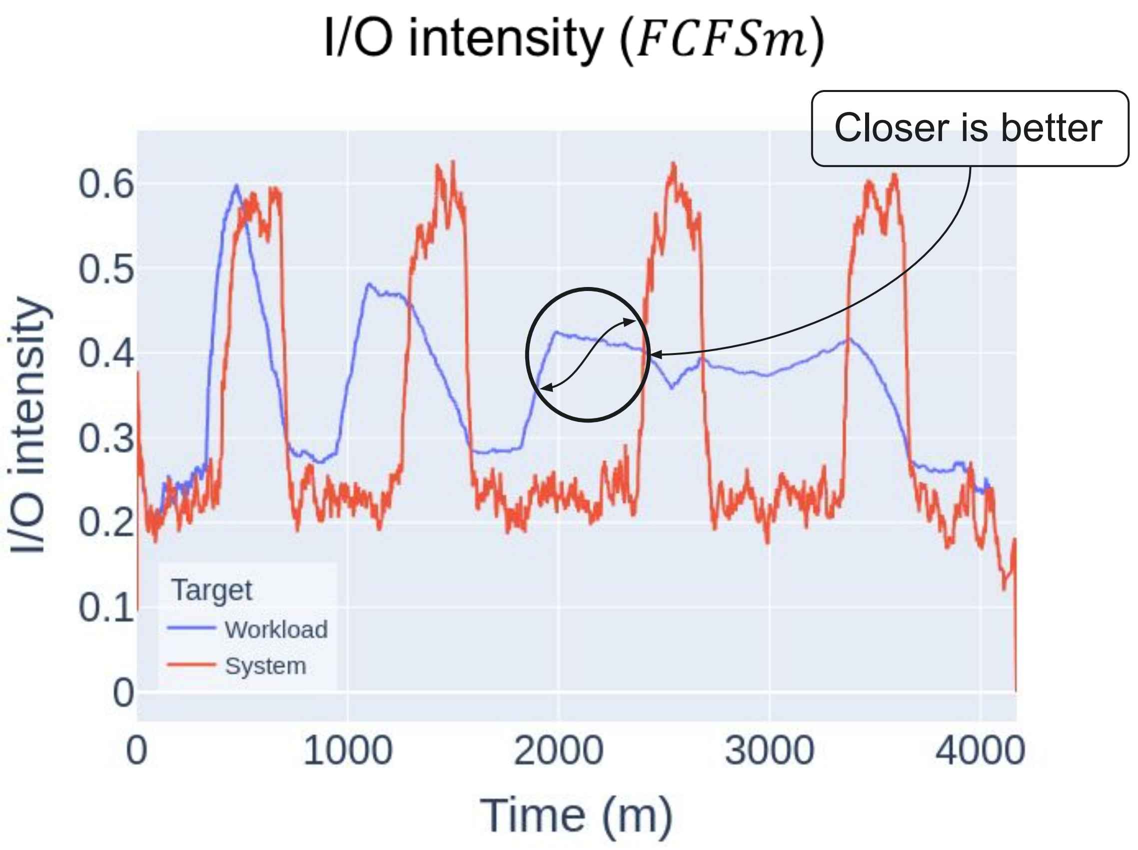
Workload	Generation parameters
Number of jobs	4000
Number of I/O peaks	4
Jobs per I/O peak	200
Computational load	
Average I/O load	
Peak I/O load	

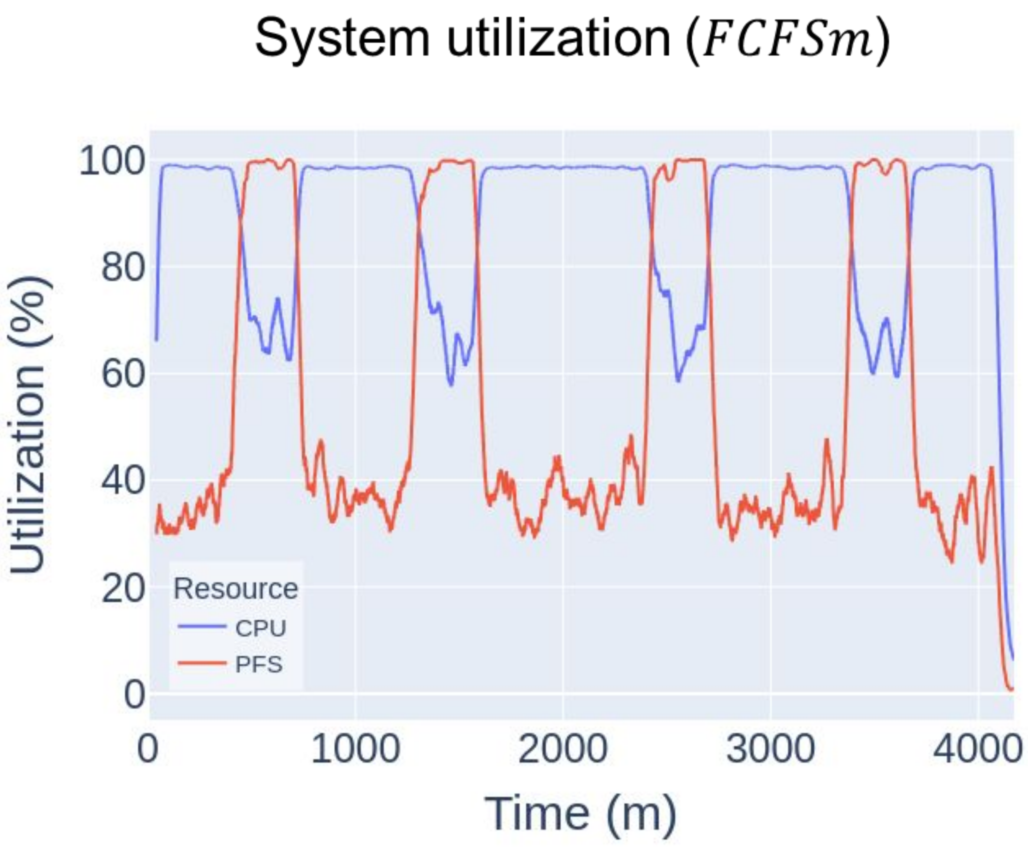
Workload generation



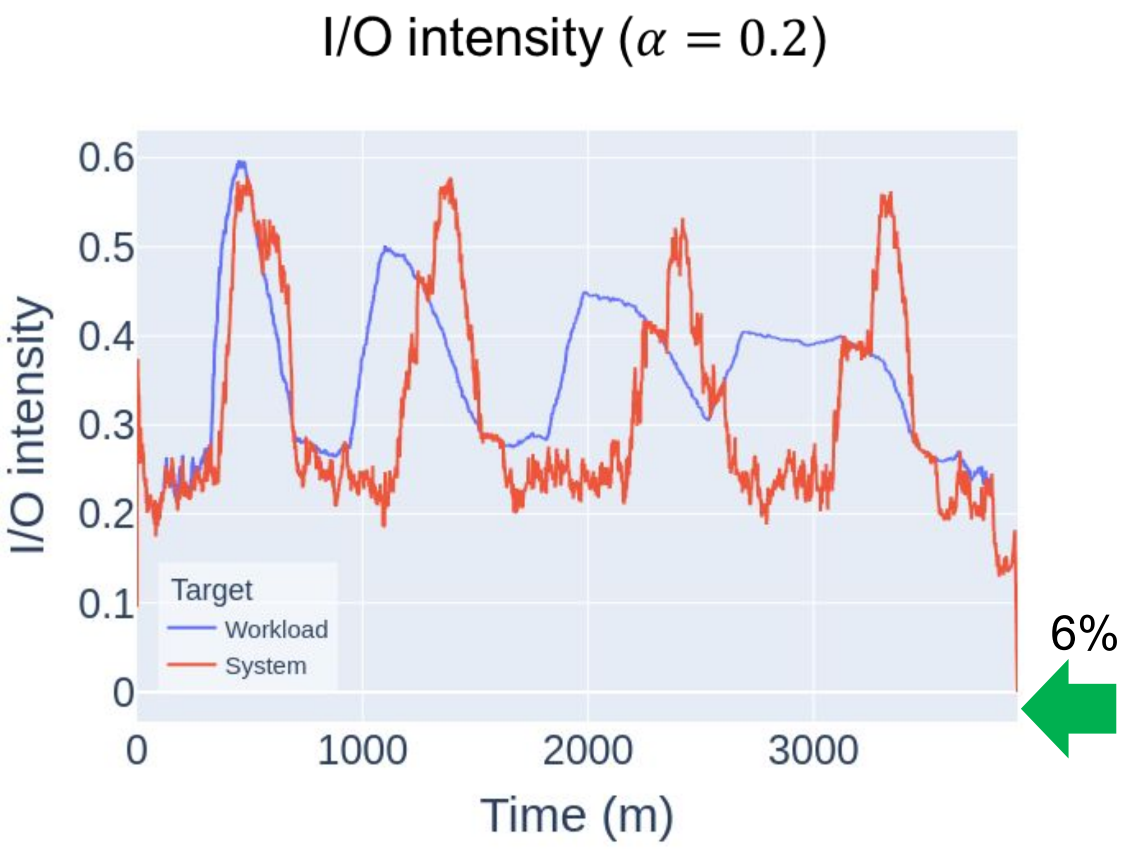


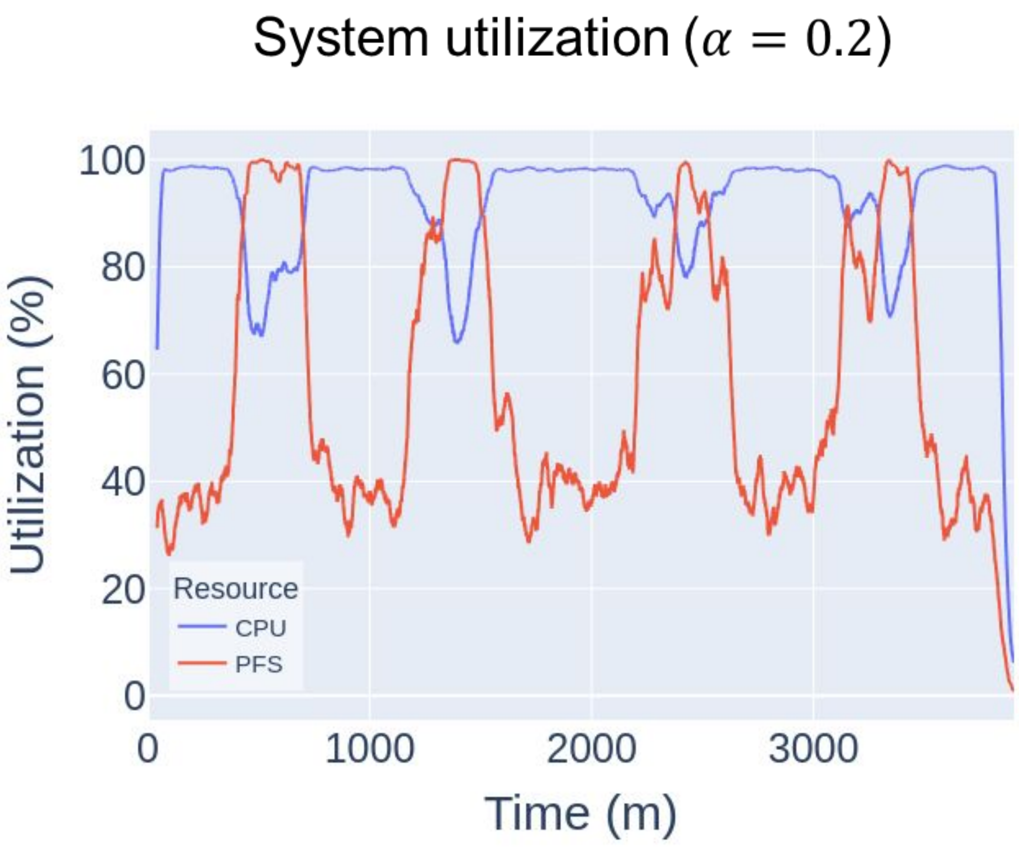
Experimental results





Experimental results

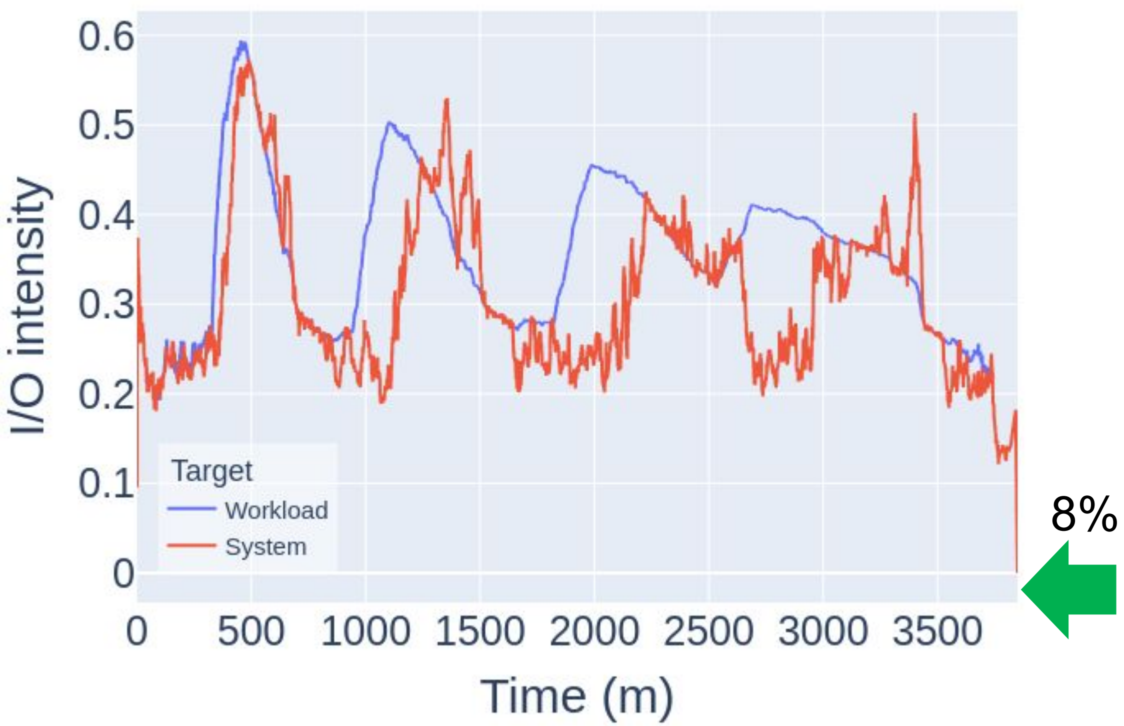




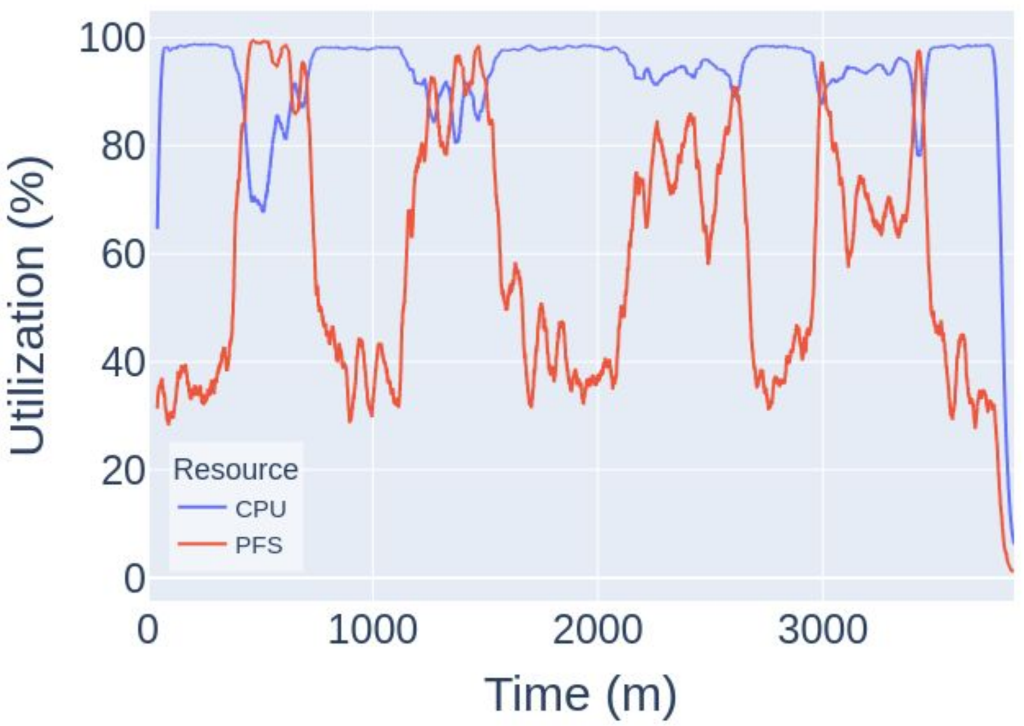
Experimental results

TECHNISCHE UNIVERSITÄT DARMSTADT
Parallel Programming

I/O intensity ($\alpha = 0.3$)

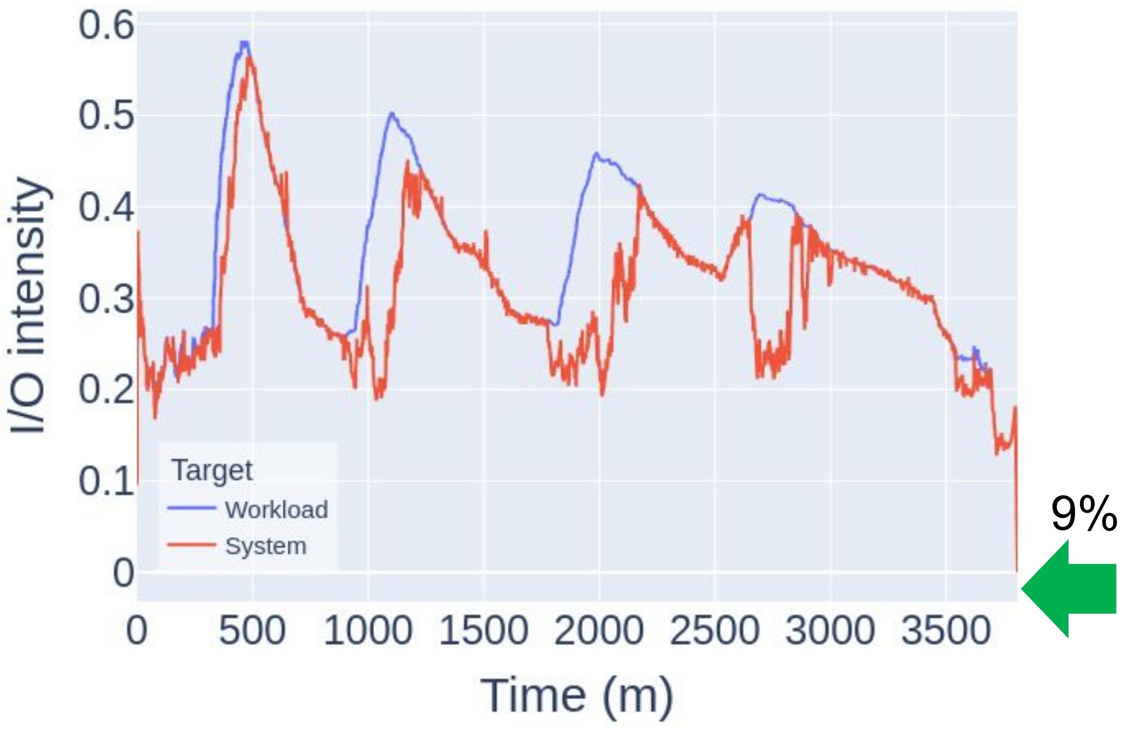


System utilization ($\alpha = 0.3$)

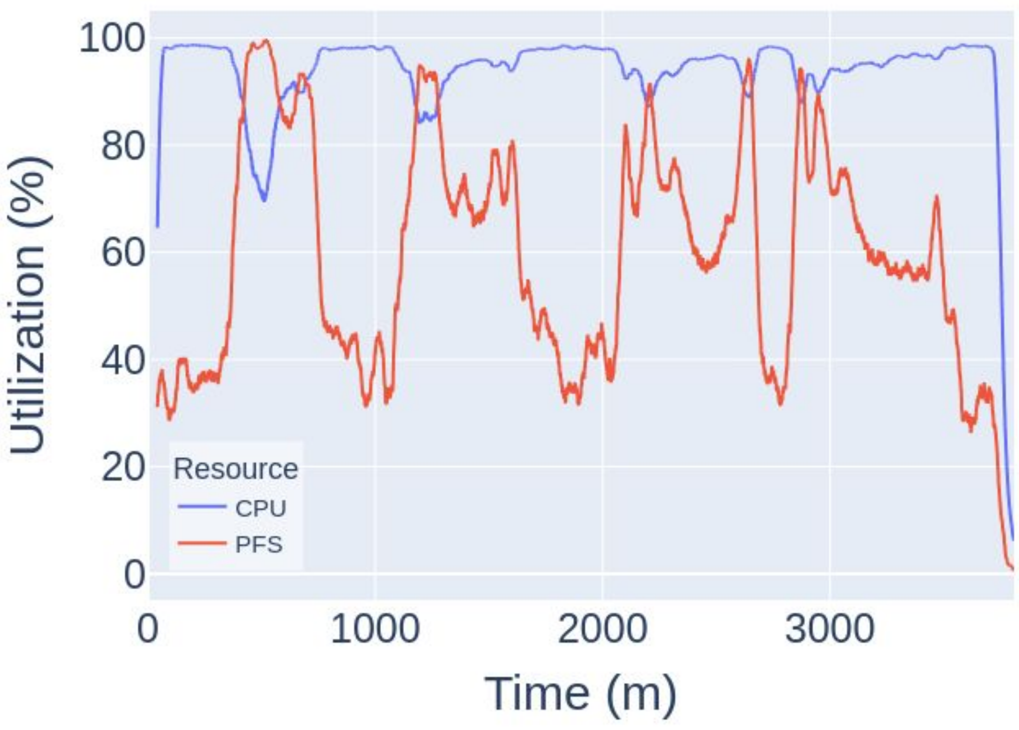


Experimental results

I/O intensity ($\alpha = 0.4$)

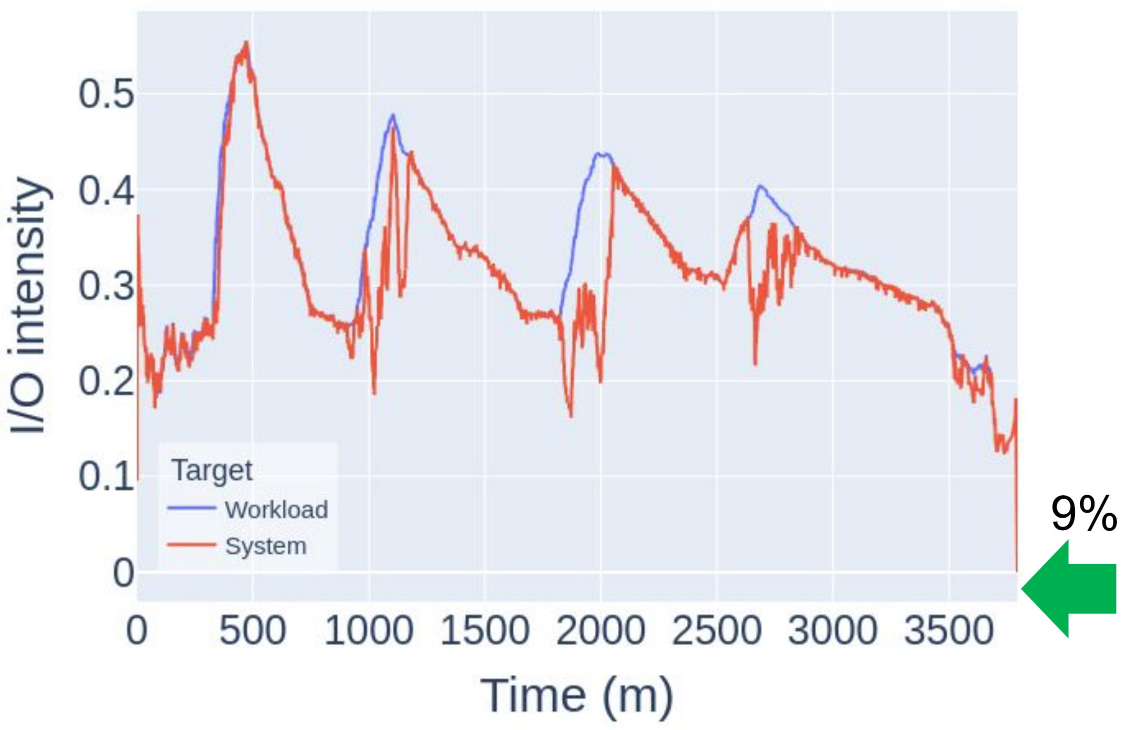


System utilization ($\alpha = 0.4$)

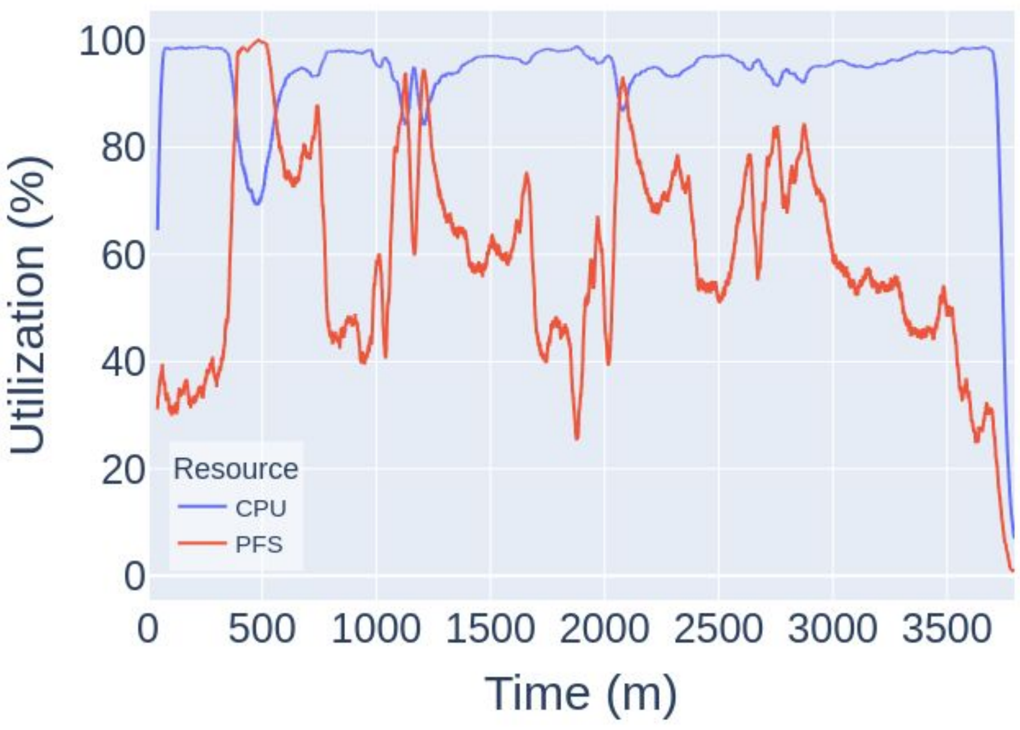


Experimental results

I/O intensity ($\alpha = 0.5$)

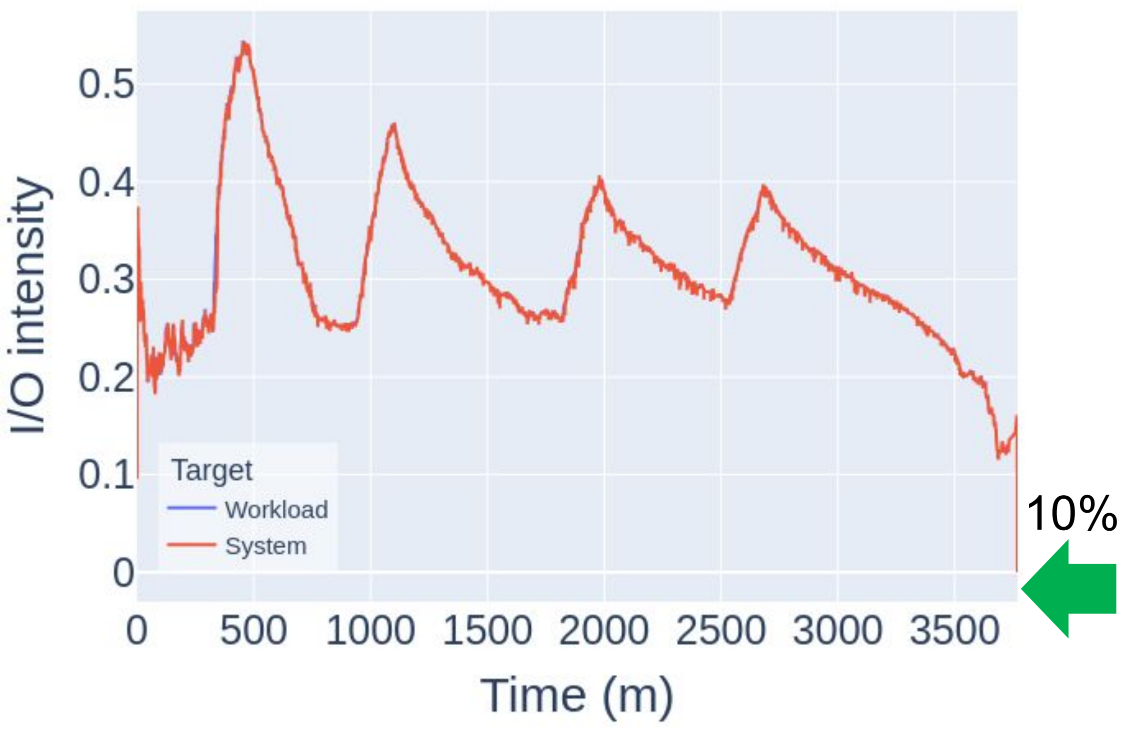


System utilization ($\alpha = 0.5$)

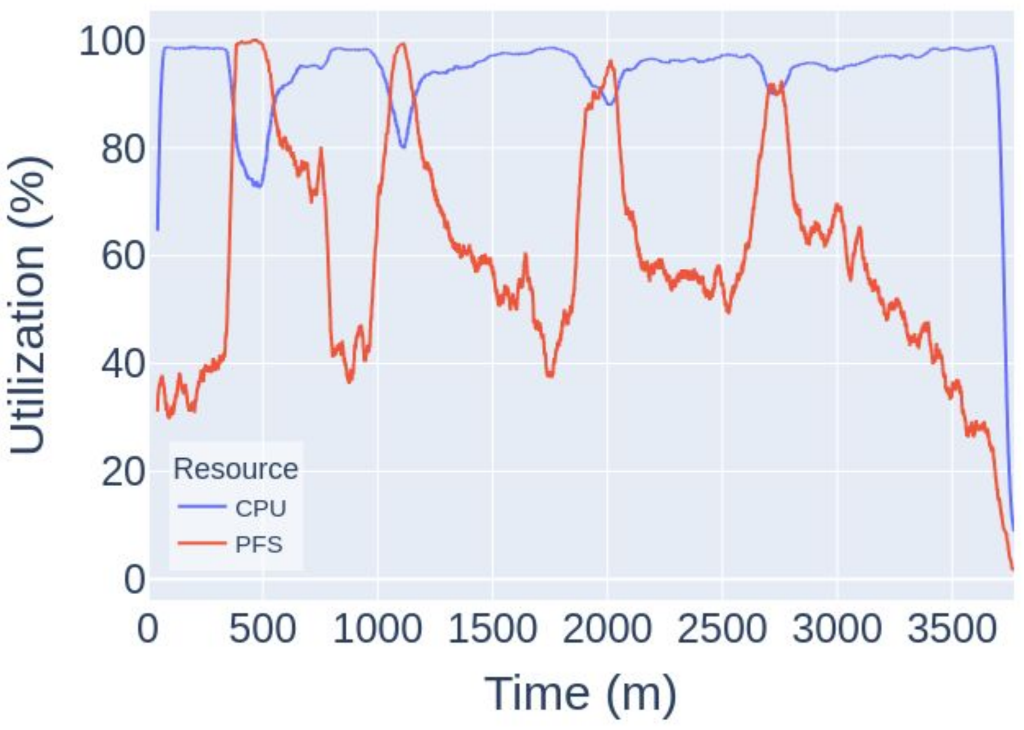


Experimental results

I/O intensity ($\alpha = 0.6$)

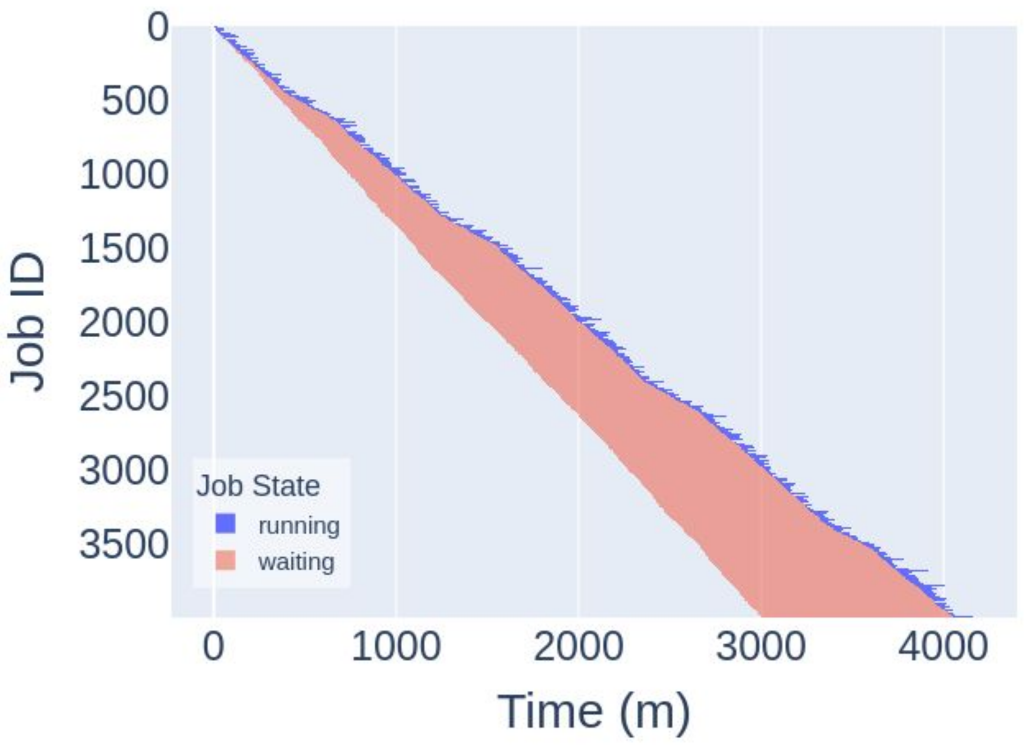


System utilization ($\alpha = 0.6$)

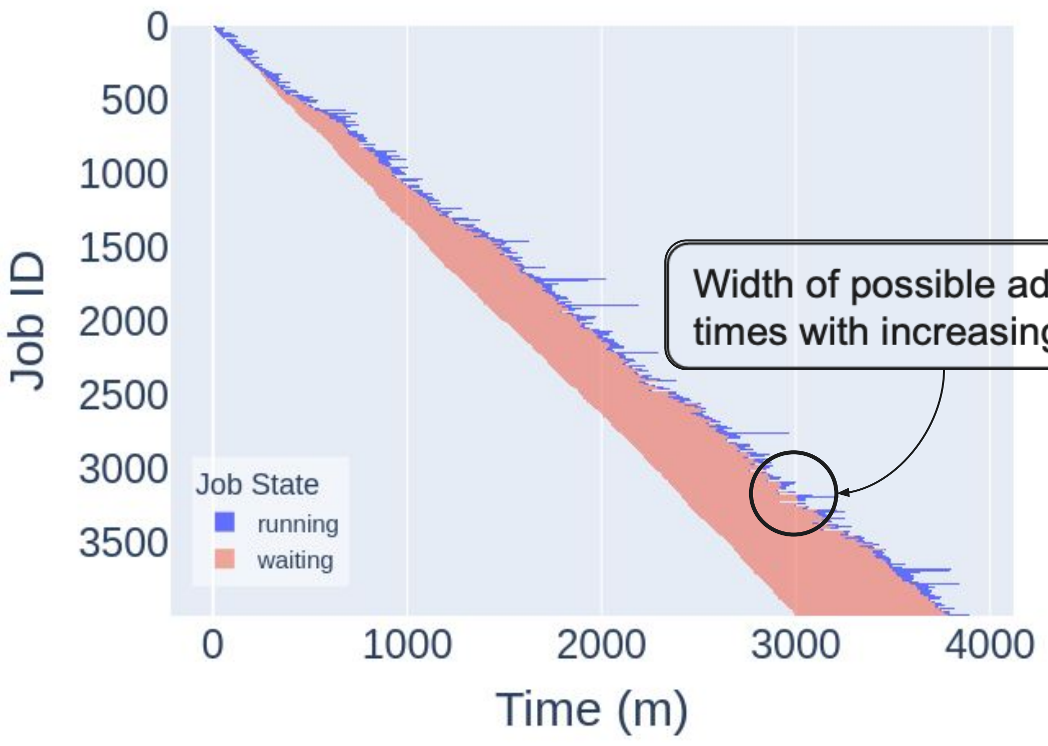


Reordering strength

$FCFSm$



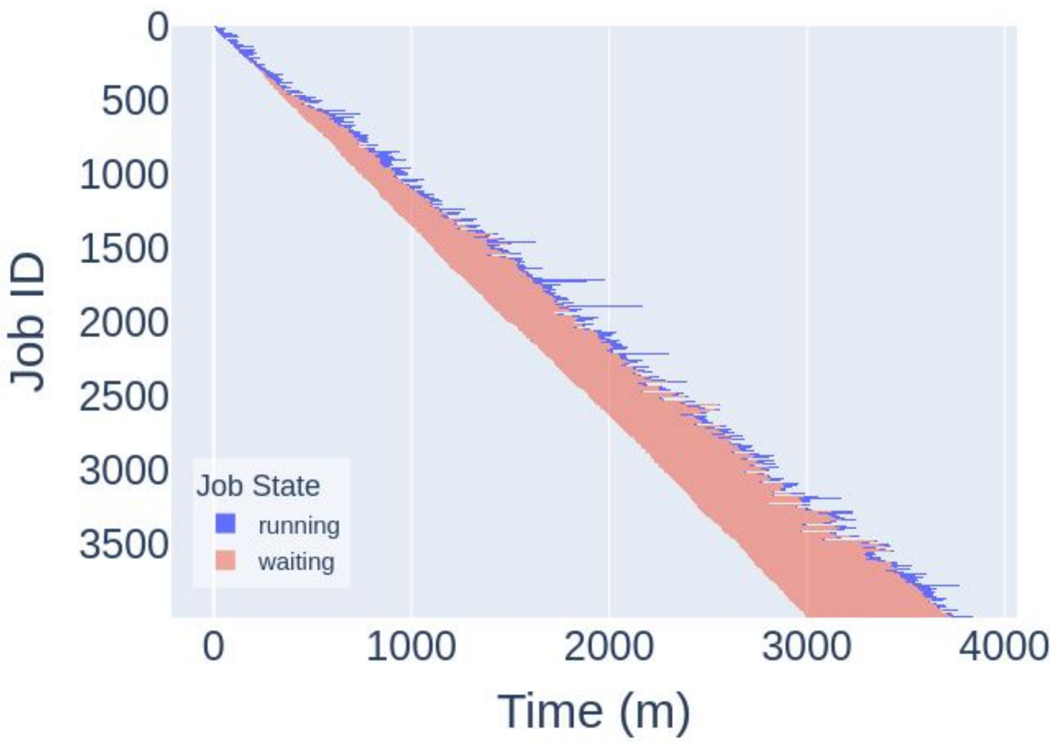
$\alpha = 0.2$



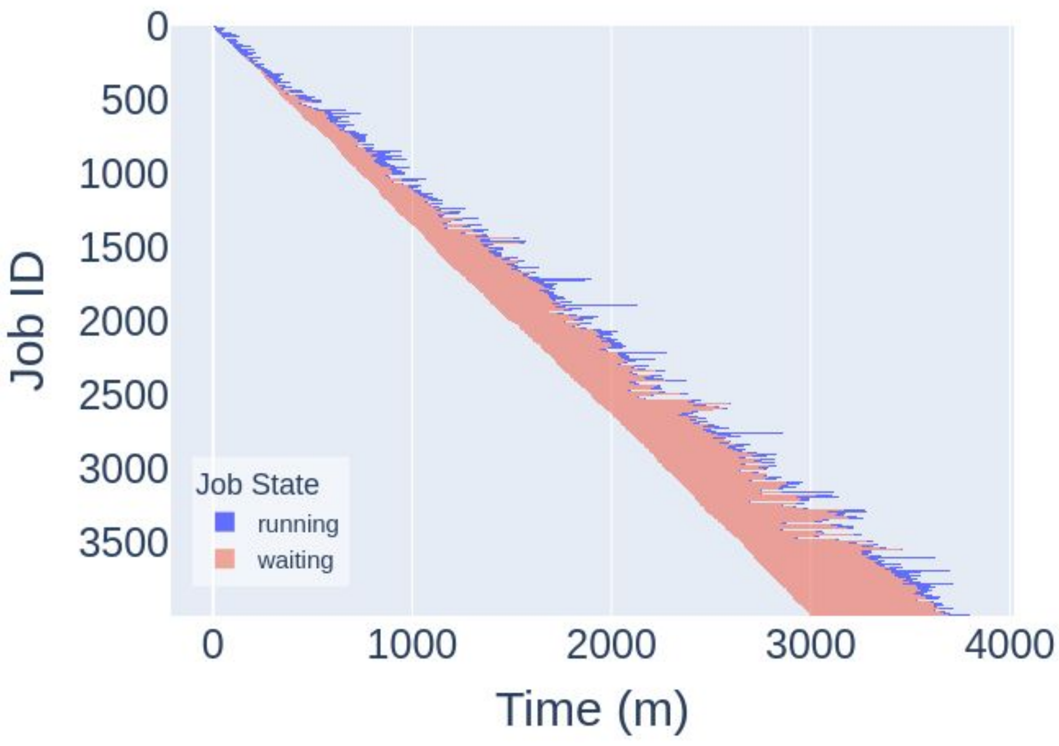
Width of possible admission times with increasing α

Reordering strength

$\alpha = 0.3$

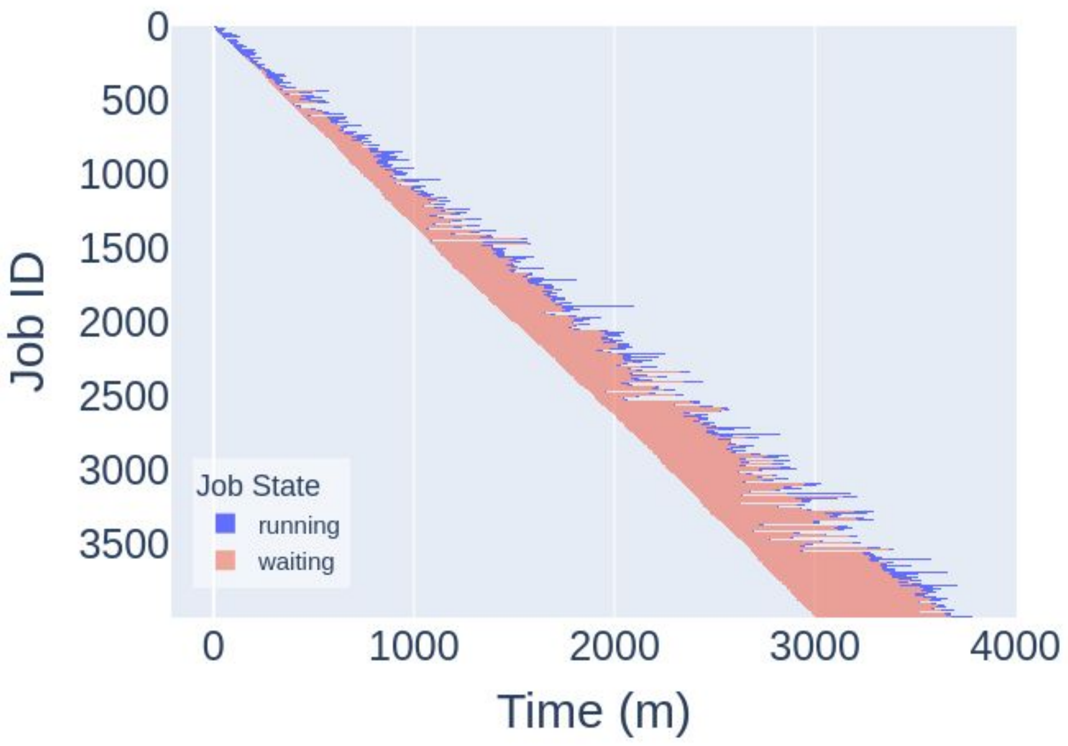


$\alpha = 0.4$

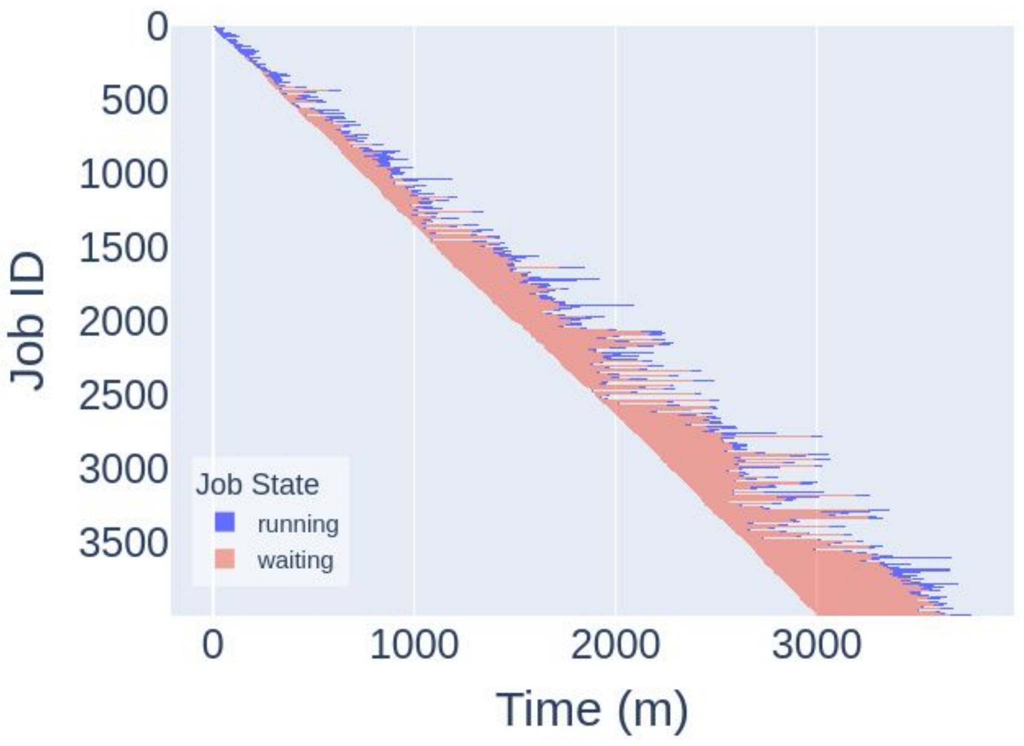


Reordering strength

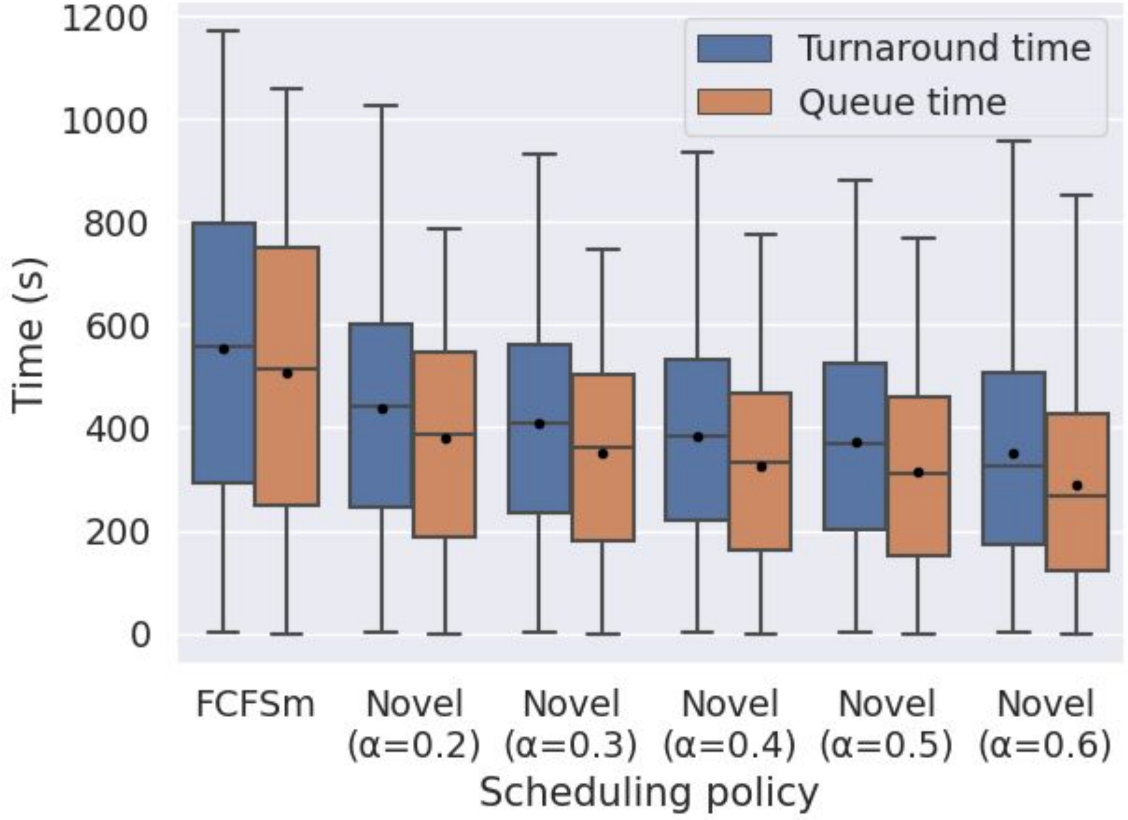
$\alpha = 0.5$

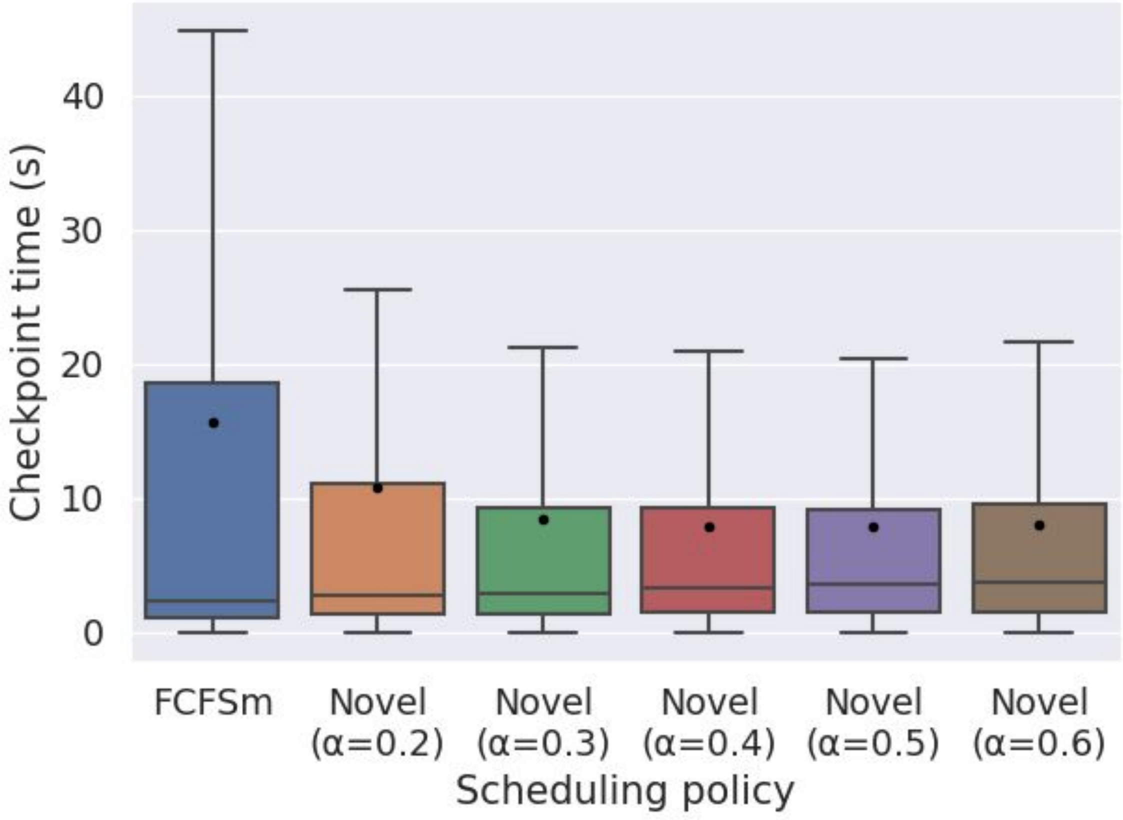


$\alpha = 0.6$



Schedule & I/O times





Conclusion & outlook

- Our approach reduces I/O time by up to 49% and makespan by up to 10%
  - For $\alpha \in [0.2{,}0.6]$
- Future work
  - Influence of I/O patterns
  - Share of jobs w/ and w/o any a-priori knowledge of I/O intensity
  - Dynamic modification of the reordering strength
  - Backfilling vs malleability

“Stealing from the rich to give to the poor”
Stealing BW from compute-intensive jobs to give it to data-intensive ones

Thank you!