Preference-Aware Scheduling for 6G Testbeds Using Matching Theory

Donatos Stavropoulos, Thanasis Korakis
CERTH, The Centre for Research & Technology Hellas, Greece
Department of Electrical and Computer Engineering, University of Thessaly, Greece

***Abstract*—6G testbeds bring together heterogeneous compute and radio resources to support cross-domain experimentation, including networking workloads that integrate AI functionality. Existing schedulers, such as backfilling, priority, and quota mechanisms, react locally to events and offer no guarantees of Pareto efficiency or preference satisfaction in non-monetary, multi-tenant environments. We adapt the *Top Trading Cycles and Chains (TTCC)* mechanism from matching theory to treat each resource release as a trigger for a global, preference-aware reallocation. Under declared QoS preferences, TTCC yields Pareto-efficient assignments while maintaining high utilization. Simulations across multiple cluster scales and load regimes show that TTCC, augmented with an EASY-style backfilling step when no further trades are possible, matches backfilling on utilization, improves preference-aware utility relative to non-priority baselines, and preserves competitive P95/P99 bounded slowdown. TTCC therefore provides a principled, preference-aware alternative to heuristic scheduling for multi-tenant 6G testbeds.**

I. Introduction

The emergence of 6G research testbeds has created multi-tenant environments where researchers deploy AI/ML workloads and networking experiments on heterogeneous and indivisible resources such as GPUs, edge nodes, and software-defined radio (SDR) nodes. Workloads typically execute either as containerized services (e.g., via Kubernetes-based GPU clusters) or as exclusive reservations in testbed control frameworks, but in all cases jobs hold dedicated resources for their execution duration. Resource availability therefore evolves dynamically as jobs arrive, start, finish early, or migrate.

Existing schedulers handle such dynamic events through queue-driven heuristics or priority mechanisms. In HPC systems, resource changes are handled by backfilling policies such as EASY [1], [2] or Conservative [3]; in Kubernetes-based AI clusters, Quality-of-Service (QoS) and priority rules determine admission and preemption; and commercial GPU orchestrators combine priority with quotas and fair-share. While effective in their domains, these approaches respond *locally* to each event and do not re-optimize the global allocation, nor do they provide per-event guarantees of Pareto efficiency or preference satisfaction in non-monetary academic environments.

We propose a new approach based on the *Top Trading Cycles and Chains (TTCC)* mechanism from matching theory [4]–[6]. Originally developed for kidney exchange, TTCC extends the well-known Top Trading Cycles (TTC) algorithm [4] by allowing *altruistic donors* to initiate chains of reallocations. We draw a natural analogy to testbed scheduling: when a job completes, its released resource acts as an altruistic donor that can trigger preference-improving exchanges among waiting jobs and, when migration is enabled, running jobs. By embedding Quality-of-Service (QoS) requirements into job preferences, TTCC yields allocations that are Pareto-efficient by construction.

This paper (i) formalizes a system model for preference-aware scheduling in heterogeneous 6G testbeds, (ii) adapts TTCC to completion-triggered global reallocations, and (iii) evaluates its performance in a discrete-event simulator using synthetic 6G workloads that model AI-enhanced networking jobs with heterogeneous GPU demands and QoS classes. TTCC provides a principled alternative to heuristic and priority-based schedulers, fitting the needs of academic testbeds that prioritize fairness and efficiency over pricing.

This work has received funding from the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe research and innovation programme, in the frame of the NATWORK project (Net-Zero self-adaptive activation of distributed self-resilient augmented services) under Grant Agreement No 101139285.

II. Background: TTC and TTCC

Matching theory offers mechanisms for allocating indivisible resources without monetary transfers. Two central algorithms are *Top Trading Cycles (TTC)* and its extension, *Top Trading Cycles and Chains (TTCC)*.

A. Top Trading Cycles (TTC)

Shapley and Scarf introduced TTC for the classic *housing market problem*, where each agent initially owns one indivisible good and ranks all goods by preference. In each round, every agent points to its most-preferred good, and each good points to its current owner; at least one directed cycle always exists. All cycles are resolved simultaneously, assigning each agent the good it points to, and the process repeats until all agents are matched. TTC yields *Pareto-efficient* and *strategy-proof* outcomes under standard assumptions.

B. Top Trading Cycles and Chains (TTCC)

Real systems often involve dynamic arrivals. In kidney exchange, for instance, *altruistic donors* enter without requiring a kidney in return. Roth *et al.* extended TTC to *Top Trading Cycles and Chains (TTCC)*, where such altruists initiate directed chains that continue as long as compatible recipients exist. TTCC retains the efficiency and incentive properties of TTC while enabling these dynamic, chain-based reallocations.

ISBN: 978-3-903176-79-9

C. Mapping TTCC to Testbed Scheduling

The analogy to testbed scheduling is direct:

- **Agents and goods:** Jobs act as agents; GPUs and other resources are the indivisible goods.
- **Preferences:** QoS and performance requirements define each job's ranking over resources.
- **Dynamic events:** When a job completes, its freed resource becomes an *altruistic donor*, initiating possible reallocation chains among waiting and running jobs.

Unlike queue-local heuristics, TTCC treats each completion event as a trigger for *global reallocation* under the declared QoS preferences. Our prior study [7] demonstrated the practicality of TTC for static, reservation-based allocation of indivisible testbed resources; here we generalize the mechanism to dynamic, event-driven scheduling with early completions and chained reallocations.

III. Related Work

This work draws upon four main research threads: (i) backfilling and reservation policies in HPC schedulers, (ii) priority-, QoS-, and quota-based mechanisms in cluster managers, (iii) matching-theoretic approaches applied to resource allocation, and (iv) migration and checkpointing for ML/HPC workloads.

a) Backfilling and reservations (HPC): EASY and Conservative backfilling are canonical HPC policies that increase utilization by permitting shorter jobs to run ahead of others in the queue, provided this does not delay any reserved start times. A reservation is the earliest time a waiting job is predicted to start once sufficient resources become free; backfilling is allowed only if the backfilled job will finish before that reservation. EASY protects the reservation of only the head-of-queue job [1], [2], whereas Conservative backfilling protects reservations for all waiting jobs [3]. Both policies treat resource releases as local queue events that admit additional waiting jobs when feasible, but once jobs are running, global reallocation is typically not reconsidered.

b) Priority/QoS, gang, quotas (cluster managers): The Kubernetes default scheduler provides priority, preemption, and class-based QoS, but it does not support batch semantics such as gang scheduling. In this context, preemption refers to evicting a running pod so that a higher-priority one can start, typically without preserving application state unless external checkpointing is available. To support AI/ML and HPC-style workloads, frameworks such as Volcano and Kueue extend Kubernetes with batch queues, gang constraints, and hierarchical quota or fair-share policies [8]–[11]. When a job completes and releases resources, these systems assign the freed capacity to the next eligible workload according to queue priority, quotas, and gang feasibility. They generally avoid multi-step reallocations across running jobs because such operations increase disruption and operational complexity. As a result, they favor predictable, local decisions rather than global reoptimization.

c) Prediction-aware, CP/RL schedulers.: Recent schedulers exploit runtime/arrival prediction, constraint programming, or reinforcement learning to improve throughput and deadline adherence [12]–[14]. While effective, they remain heuristic or learned policies without matching-theoretic guarantees such as per-event Pareto efficiency.

d) Matching theory in systems: Top Trading Cycles (TTC) [4] and its extension TTCC [5], [6] provide Pareto-efficient and strategy-proof allocations under classical assumptions. Adaptations have appeared in resource exchange and datacenter placement, but prior work has not applied *matching-theoretic reallocation* to GPU/edge testbeds with heterogeneous resources and QoS preferences. TTCC extends these ideas to dynamic, event-driven scheduling through global, preference-aware reallocations.

e) Migration and checkpointing: Job migration requires either checkpoint/restart or container-level eviction, depending on the execution model used by the testbed. Mechanisms exist for both HPC and ML workloads but incur overheads that must be budgeted [15]–[17]. Practical systems therefore restrict migration frequency or scope, motivating our deployable variant that also operates in a *waiting-only* mode when migration support is limited or unavailable.

f) Fair and matching-inspired scheduling: A complementary line of work targets fairness and Pareto-leaning outcomes in multi-tenant clusters. Dominant Resource Fairness (DRF) [18] provides a foundational notion of proportional fairness and underlies many quota/fair-share schedulers. For GPU clusters, systems such as Gandiva and Tiresias improve job-level fairness via time-slicing and queue management [19], [20]. Syrigos *et al.* introduced MLAS, an MLOps framework that optimizes ML workflow placement across Cloud and Edge [21], and later proposed EELAS, an energy-efficient and latency-aware scheduler for cloud-native ML workloads [22]. These approaches share our motivation for resource-aware and latency-sensitive allocation, but differ in mechanism: they enhance scheduling heuristics or infrastructure orchestration, whereas TTCC performs *event-driven, global* reallocations via preference graphs and exchange chains, yielding *per-event Pareto-efficient* outcomes under declared preferences.

Across schedulers, a resource-release or completion event produces a scheduling tick. Conventional policies handle these events locally: backfilling and quota systems admit the next feasible waiting jobs, while running jobs remain fixed. TTCC instead treats each such event as the start of a *cluster-wide* reallocation phase over all nodes in the testbed. Relative to existing mechanisms, TTCC differs along four dimensions: (i) its **scope** (cluster-wide reoptimization rather than local queue fit), (ii) its **process** (multi-step chains or cycles rather than one-step greedy advancement), (iii) its **preference awareness** (explicit rankings over heterogeneous resources), and (iv) its **outcomes** (per-event Pareto efficiency under declared preferences).

IV. System Model

We consider a 6G research testbed with heterogeneous, indivisible resources (e.g., GPUs, SDRs, edge nodes). Each resource unit can be allocated to at most one job at a time. Jobs arrive dynamically; some finish before their reserved time and release resources early.

A. Jobs

The workload is $J = \{j_1, \ldots, j_n\}$. Each job j is characterized by:

- *Arrival time* a_j, *start time* s_j, and *completion time* c_j
- *Demand* $d_j \in \mathbb{N}$ resource units
- *Reserved duration* r_j and *actual runtime* $t_j = c_j - s_j \leq r_j$
- *Deadline* D_j
- *QoS class* $q_j \in \{\text{Gold, Silver, Bronze}\}$

When $t_j < r_j$, job j releases its resources early, enabling TTCC to initiate chain reallocations.

B. Resources and Preferences

Let $R = \{r_1, \ldots, r_m\}$ denote the set of resource units (e.g., edge and cloud GPUs). Each job j defines a strict ranking $\rho_j(\cdot) \in \{1, 2, \ldots\}$ over feasible resources, where $\rho_j(r)$=1 denotes the most preferred (top-choice) unit, $\rho_j(r)$=2 the next, and so on. If $A(j) = \{r_{j,1}, \ldots, r_{j,d_j}\}$ is the allocation bundle for j, the corresponding *preference satisfaction* is

$$S_j = \frac{1}{d_j} \sum_{k=1}^{d_j} \gamma^{\,\rho_j(r_{j,k})-1}, \qquad \gamma \in (0, 1]. \tag{1}$$

so that S_j=1 when all allocated units are top choices, decreasing geometrically with lower-ranked resources.

C. Per-Job Utility

Each job's utility depends jointly on timeliness, QoS class, and allocation preferences. Let the *tardiness* be

$$T_j = \max(0,\, c_j - D_j),$$

where D_j is the deadline. For each QoS class $q_j \in \{\text{Gold, Silver, Bronze}\}$, we assign a priority weight w_{q_j} and a decay scale β_{q_j} satisfying

$$w_{\text{Gold}} > w_{\text{Silver}} > w_{\text{Bronze}}, \qquad \beta_{\text{Gold}} < \beta_{\text{Silver}} < \beta_{\text{Bronze}}.$$

The base (deadline-aware) utility is

$$U_{\text{base}}(j) = w_{q_j} \exp\Big(-\frac{T_j}{\beta_{q_j}}\Big), \tag{2}$$

which attains full utility w_{q_j} when $c_j \leq D_j$ and decays exponentially with increasing lateness.

Preference satisfaction scales this value, giving the overall per-job utility

$$U_j = U_{\text{base}}(j)\, S_j. \tag{3}$$

Thus, deadline adherence and allocation quality jointly determine the effective reward perceived by each job.

D. Evaluation Metrics

We evaluate scheduling performance using three complementary metrics that capture resource efficiency, user satisfaction, and responsiveness.

a) Utilization: Utilization measures the fraction of total resource capacity actively used during the observation window. Let $J_{\text{fin}} = \{\, j \in J : c_j \text{ is defined} \,\}$ be the set of completed jobs, and define the *makespan window* as

$$T_{\text{MS}} = \max\Big(1,\, \max_{j \in J_{\text{fin}}} c_j - \min_{j \in J_{\text{fin}}} a_j\Big),$$

which spans from the first observed arrival to the last completion. With M total resource units, utilization is

$$\text{Utilization} = \frac{\sum_{j \in J_{\text{fin}}} d_j\, t_j}{M\, T_{\text{MS}}}. \tag{4}$$

A value of 1 indicates fully occupied resources throughout the makespan window, whereas smaller values reflect idle capacity.

b) Average Utility: The system-wide preference-aware performance is expressed as the average per-job utility:

$$\bar{U} = \frac{1}{|J_{\text{fin}}|} \sum_{j \in J_{\text{fin}}} U_j, \tag{5}$$

where U_j is defined in (3). This metric combines QoS compliance, deadline adherence, and preference satisfaction.

c) Bounded Slowdown (Tail): To assess responsiveness across workloads, we use the bounded slowdown metric, defined for each completed job as

$$\text{BSLD}_j = \frac{R_j}{\max(t_j, \tau)}, \qquad R_j = c_j - a_j, \tag{6}$$

where R_j is the response time and τ is a small runtime floor (e.g., one minute) to prevent inflation for short jobs. We summarize the distribution by its high-percentile tails,

$$\text{P95} = Q_{0.95}(\text{BSLD}), \qquad \text{P99} = Q_{0.99}(\text{BSLD}), \tag{7}$$

where $Q_p(\text{BSLD})$ denotes the p-quantile over $\{\text{BSLD}_j\}_{j \in J_{\text{fin}}}$.

V. Proposed Scheduler: TTCC Adaptation

We adapt *Top Trading Cycles and Chains (TTCC)* to the domain of 6G testbed scheduling. The key idea is to interpret *each resource-release event* (typically when a job completes) as a trigger for constructing allocation chains. These chains reassign both waiting and, when allowed, currently running jobs in a way that is Pareto-efficient under QoS preferences.

A. Algorithm Overview

The scheduler operates as follows:

1) **Preference construction:** Each job j maintains a strict ranking over feasible resources R, derived from its QoS class and deadline (e.g., Gold jobs rank low-latency edge GPUs above cloud GPUs). Ties are broken deterministically.
2) **Graph building:** We construct a directed graph on jobs and resources:

- Each *resource* points to the highest-priority (QoS-weighted), compatible job that can use it.
- Each *waiting job* points to its most-preferred feasible resource.
- Each *running job* points to its *currently held* resource (its "paired donor").

3) **Cycle resolution (TTC):** If directed cycles exist, execute them simultaneously. Each job in a cycle receives the resource it points to; the involved jobs/resources are removed from the graph.
4) **Chain resolution (TTCC):** When a resource becomes free (e.g., upon a completion), it acts as an altruistic donor and initiates a chain that alternates

$$r \to j \to r' \to j' \to \cdots$$

where $r \to j$ is the freed (or subsequent) resource pointing to its most-preferred compatible job, and $j \to r'$ is: (i) the job's *current* resource if j is running, or (ii) **terminal** if j is waiting (no current resource). The chain stops when it reaches a waiting job (proper chain) or revisits a node (forming a cycle).
5) **Reallocation:** Execute all detected cycles and any proper chains by shifting assignments along each path (each job receives the predecessor resource). When migration is enabled (e.g., with checkpoint/restart), running jobs may move; otherwise only waiting jobs are reallocated.

After all TTCC chains and cycles are exhausted at a completion event, some resources may still remain idle. To keep the system work-conserving, the scheduler then performs an EASY-style backfilling pass over the remaining waiting jobs: it admits any job that fits without violating reservations, but never migrates running jobs. This hybrid design preserves TTCC's Pareto-efficient reallocations while recovering the high utilization of classical backfilling.

B. Example Walk-through

Consider two GPUs with distinct characteristics:

$$r_{\text{edge}} \text{ (low latency)}, \qquad r_{\text{cloud}} \text{ (high throughput)}.$$

Jobs have simple preferences:

$$C_{\text{Gold}} : r_{\text{edge}} \succ r_{\text{cloud}},$$
$$A_{\text{Silver}},\ B_{\text{Bronze}} : r_{\text{cloud}} \succ r_{\text{edge}}.$$

Initial state. Bronze job B_{Bronze} runs on r_{edge}, Silver job A_{Silver} runs on r_{cloud}, and Gold job C_{Gold} is waiting:

$$r_{\text{edge}} \to B_{\text{Bronze}}, \qquad r_{\text{cloud}} \to A_{\text{Silver}}, \qquad C_{\text{Gold}} \to \text{waiting}.$$

Completion and reallocation. When A_{Silver} completes, r_{cloud} becomes free and TTCC initiates a reallocation chain:

$$r_{\text{cloud}} \to B_{\text{Bronze}} \to r_{\text{edge}} \to C_{\text{Gold}}.$$

Job B_{Bronze} moves from r_{edge} to its preferred resource r_{cloud}, freeing r_{edge} for C_{Gold}, which starts immediately.

Outcome.

$$r_{\text{edge}} \to C_{\text{Gold}}, \qquad r_{\text{cloud}} \to B_{\text{Bronze}}.$$

Both jobs improve utility: C_{Gold} runs on low-latency edge, and B_{Bronze} moves to high-throughput cloud. The reallocation is Pareto-efficient and triggered by a single completion event.

C. Complexity

Let n denote the number of jobs and m the number of resources. Constructing the preference graph requires $O(n \cdot m)$ comparisons. Each cycle or chain resolution runs in $O(n)$ time, and at least one job is removed per iteration. The overall complexity is $O(n^2)$ per reallocation event, which is tractable for testbeds with hundreds of jobs.

D. Integration with Existing Schedulers

TTCC complements rather than replaces existing batch systems. It can be invoked *on resource-release events* where backfilling or priority queues typically act locally. For example:

- **Slurm:** TTCC can be triggered as a plugin when a job completes and releases resources.
- **Kubernetes:** TTCC can be implemented as an admission controller that re-evaluates allocations upon pod termination.

This integration enables principled, global reallocations without modifying the base scheduler.

E. Waiting-Only Variant Used in Experiments

The full TTCC mechanism can, in principle, migrate running jobs by including them in the preference graph and executing chains that move their allocations. In this paper we evaluate a restricted, deployable variant that operates in *waiting-only* mode: running jobs remain pinned to their current resources until completion, and only waiting jobs participate in TTCC chains and cycles. This design avoids the need to budget checkpoint/restart overheads while still exploiting early completions to improve preference satisfaction and latency for queued jobs.

VI. Simulation Framework

We built a discrete-event simulator that advances to the next arrival or completion event and applies the scheduling policy active at that moment. TTCC performs a cluster-wide, preference-aware reallocation whenever a job completes, considering all *waiting* jobs and all resources in the cluster. Running jobs never migrate: once started, they retain their resources until completion. When no further TTCC exchanges are possible at an event, the scheduler invokes an EASY-style backfilling step to admit any additional waiting jobs that fit, ensuring the cluster remains work-conserving.

The baseline schedulers act locally: they handle each event by updating their queues and admitting the next feasible waiting jobs without reconsidering the placement of running jobs. The only exception is the Priority/QoS policy, which may preempt lower-priority running jobs to admit higher-priority

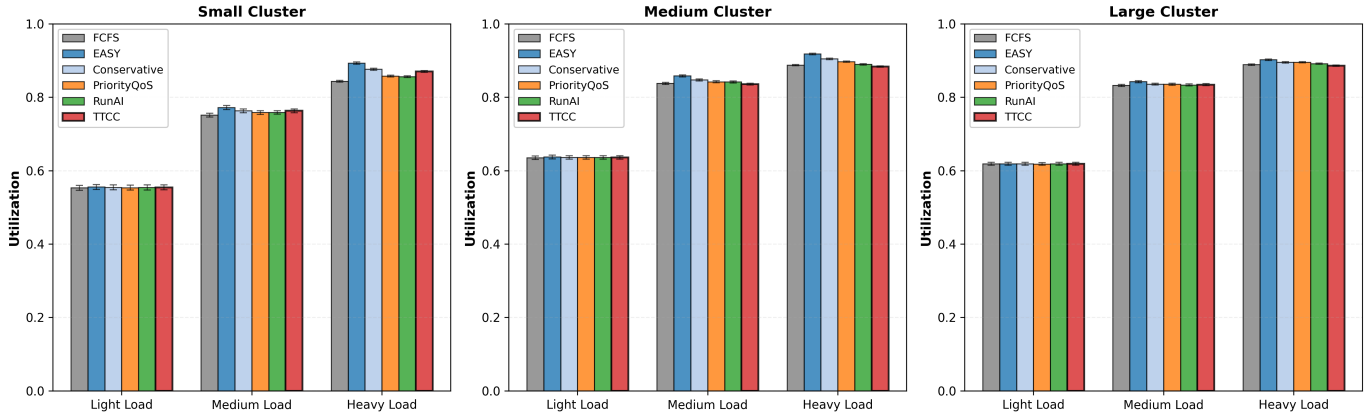

Fig. 1. Cluster utilization versus offered load, averaged over cluster sizes.

TABLE I
WORKLOAD PARAMETERS USED IN THE SIMULATIONS.

Parameter	**Value**
Arrival process	Poisson (λ)
GPU demand (Small)	$\{1, 2, 3, 4\}$ with $(.40, .35, .20, .05)$
GPU demand (M/L)	$\{1, 2, 3, 4, 8\}$ with $(.35, .30, .20, .10, .05)$
Reserved duration r_j	Exp(μ), mean 2 h
Early completion	p_{early}=0.3, $U\sim$Unif$[0.3, 0.8]$
QoS mix	Gold/Silver/Bronze = 20/50/30%
Deadlines	$D_j = a_j + \alpha_{q_j} r_j$
Deadline multipliers	$\alpha = (1.0, 1.5, 3.0)$
Utility weights w_q	$(3, 2, 1)$
Utility decay β_q	$(1, 1.5, 3)$
Preference decay γ	0.9

ones; in our simulation, such evictions incur no migration or restart cost, effectively modeling cost-free preemption. Gang semantics are enforced for multi-GPU jobs.

a) Cluster setup: We model heterogeneous 6G testbeds with edge and cloud GPUs, each an indivisible unit. Three cluster sizes are used: *Small* (8 GPUs: 4 edge, 4 cloud), *Medium* (32 GPUs: 16, 16), and *Large* (64 GPUs: 32, 32). The hardware inventory is fixed across schedulers.

b) Workload and parameterization: Job arrivals follow a Poisson process with rate λ. Each job requests a number of GPUs drawn from a discrete distribution, and its reserved duration r_j is exponentially distributed with mean 2 h. A fraction of jobs finish early according to the early-completion model described in Section IV, and all completions trigger reallocation events. QoS classes follow a 20%/50%/30% Gold/Silver/Bronze mix, with deadlines and utilities derived from the class-specific parameters. For Medium and Large clusters, we include a small proportion of 8-GPU jobs to emulate large training workloads and increase gang-allocation pressure. Table I summarizes all workload parameters.

These parameters are chosen to reflect realistic workloads encountered in GPU-enabled research testbeds. The GPU-demand distributions capture the dominance of small and medium AI/ML jobs together with a small fraction of large gang jobs that increase allocation pressure. Early completions model the well-documented tendency of users to overestimate runtime in both HPC and ML clusters. The QoS mix and deadline multipliers represent heterogeneous latency sensitivity typical of AI-enhanced networking experiments. Finally, the utility and preference parameters follow standard decay formulations used in prior scheduling and resource-allocation studies. Overall, this configuration provides diverse yet controlled workloads that enable consistent and meaningful comparison across schedulers.

c) Offered load and horizon: We define offered load as

$$L = \frac{\lambda \, \mathbb{E}[d] \, \mathbb{E}[t]}{M},$$

with M GPUs. We tune λ to target *Light*, *Medium*, and *Heavy* regimes (approximately $L \in \{0.6, 0.9, 1.2\}$ for Small/Medium/Large clusters). Each run simulates a fixed arrival window and then continues with a *drain* period until the system empties, ensuring well-defined completion times and utilization windows.

d) Protocol, metrics, and reporting: From per-job records $(a_j, s_j, c_j, A(j))$ we compute utilization, average utility $\bar{U}$, and bounded slowdown (P95/P99) as defined in Section IV. All schedulers are evaluated on identical job traces and identical sequences of random draws using *Common Random Numbers (CRN)*, a standard variance-reduction technique in simulation that ensures differences between schedulers arise from the policy rather than randomness. Results aggregate over 1000 independent CRN seeds, yielding tight confidence bounds and high statistical power. We report 95% confidence intervals (CIs) for all mean metrics and for the median slowdown percentiles. [1]

VII. PERFORMANCE EVALUATION

We compare TTCC against five representative schedulers: FCFS (no backfilling), EASY and Conservative backfilling, a Kubernetes-style Priority/QoS policy, and a RunAI-like quota/fair-share scheme. We follow the experimental protocol

[1]All simulation code, configuration files, and analysis scripts are publicly available at https://github.com/dostavro/ttcc-artifact.

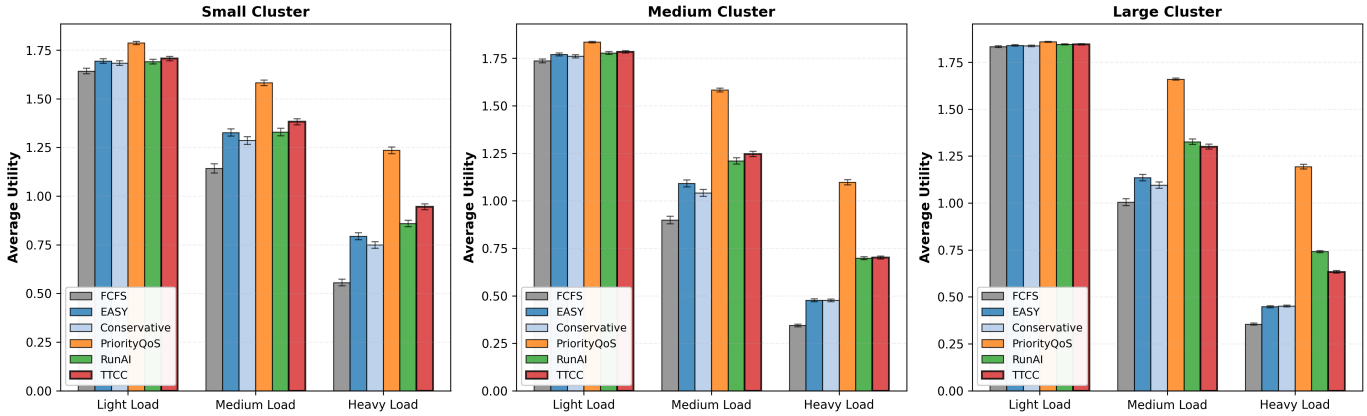

Fig. 2. Average preference-aware utility by scheduler and load.

of Section VI and use the metrics defined in Section IV. Figures 1–3 and Table II summarize the results.

A. Resource Efficiency

Across all cluster sizes and load regimes, TTCC sustains utilization comparable to the two backfilling baselines (Fig. 1). At light load, all schedulers reach similar occupancy: there is sufficient capacity and the choice of policy has little effect. Under medium and heavy load, small differences emerge. Across clusters, all evaluated schedulers form a very tight utilization band, with differences being small and within overlapping confidence intervals.

The aggregate numbers in Table II confirm this trend. TTCC attains a mean utilization of 0.765, statistically indistinguishable from EASY (0.777) and Conservative (0.770) due to overlapping confidence intervals. These results show that TTCC preserves the high efficiency of backfilling while adding preference-aware reallocations on completion events.

B. Preference-Aware Performance

Fig. 2 reports the QoS-weighted average utility across clusters and load levels. At light load, all schedulers deliver similarly high utility because most jobs complete before their deadlines on preferred resources. As load increases, differences between policies become more evident.

Across medium and heavy load, TTCC consistently outperforms the non-priority backfilling baselines (FCFS, EASY, Conservative) in all cluster sizes. The gains are modest at medium load and grow under heavier contention, reflecting TTCC's ability to improve allocation quality for waiting jobs without disrupting running ones. RunAI tracks TTCC closely and in some cases slightly exceeds it, whereas Priority/QoS achieves the highest utility overall because it may freely evict lower-priority jobs without incurring any migration or restart costs in our simulation model. Such cost-free preemption amplifies its advantage but does not reflect the overheads present in real systems.

Aggregated across all scenarios (Table II), TTCC improves mean utility by about 20% over FCFS and roughly 10–12% over EASY and Conservative, while slightly exceeding RunAI. These results indicate that TTCC delivers stronger preference-aware performance than non-priority baselines while avoiding the aggressive, cost-free preemption relied upon by Priority/QoS.

C. Tail Responsiveness

Fig. 3 shows the bounded slowdown percentiles (P95/P99). Conservative backfilling and TTCC provide the tightest extreme tails among the evaluated schedulers.

TTCC achieves tail performance very close to Conservative across cluster sizes and load levels. Because the evaluated variant does not migrate running jobs and falls back to EASY-style admission after trading, waiting jobs retain similar protection from starvation. At light load, all schedulers exhibit small slowdowns, while under medium and heavy load TTCC remains within a narrow margin of Conservative and clearly outperforms FCFS, RunAI, and Priority/QoS in P95 and P99 tails.

D. Utility–Latency Trade-off and Aggregate Comparison

The aggregate utility–latency trade-off is summarized in Fig. 4, which plots each scheduler's mean utility against its mean P99 bounded slowdown across all nine scenarios. TTCC lies near the upper-left frontier: it delivers clearly higher utility than FCFS, EASY, and Conservative, and slightly higher utility than RunAI, while maintaining a P99 slowdown close to the best backfilling policy. Priority/QoS achieves the highest utility overall, but at the cost of P99 slowdowns more than three times those of TTCC.

Table II quantifies these trends. TTCC's mean utility (1.283) exceeds all non-priority baselines and trails Priority/QoS by a moderate margin. Its P95 and P99 slowdowns (12 and 31) are slightly better than or comparable to Conservative's (16 and 31), while remaining far better than FCFS, Priority/QoS, and RunAI. In aggregate, TTCC occupies a favorable operating point on the utility–latency Pareto frontier, providing a principled middle ground between heuristic backfilling and rigid priority enforcement.

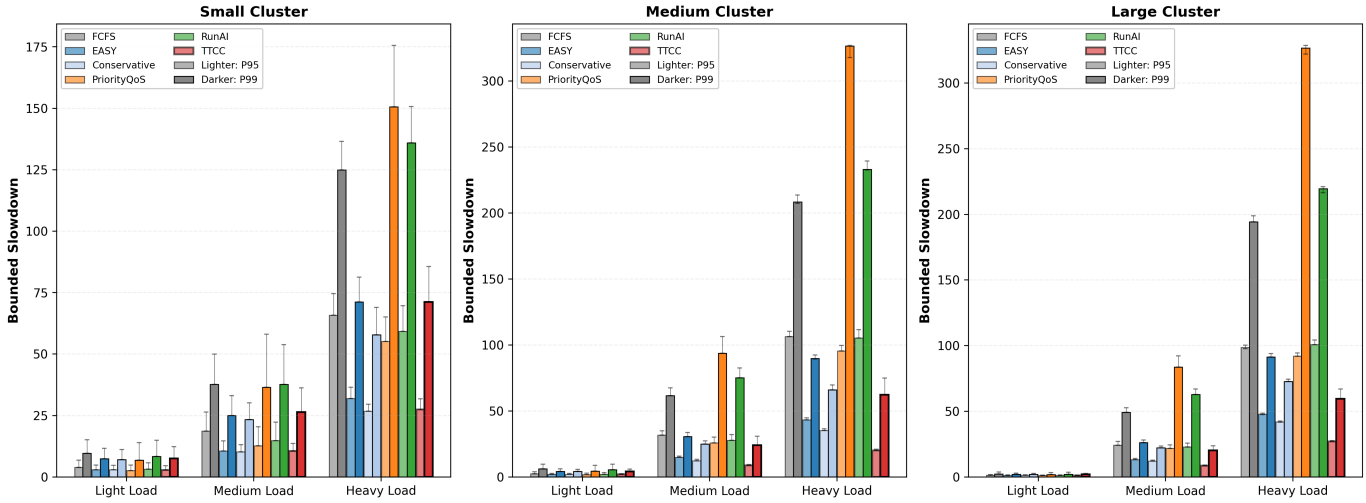

Fig. 3. Bounded slowdown tails (P95/P99) by scheduler and load. Lower is better.

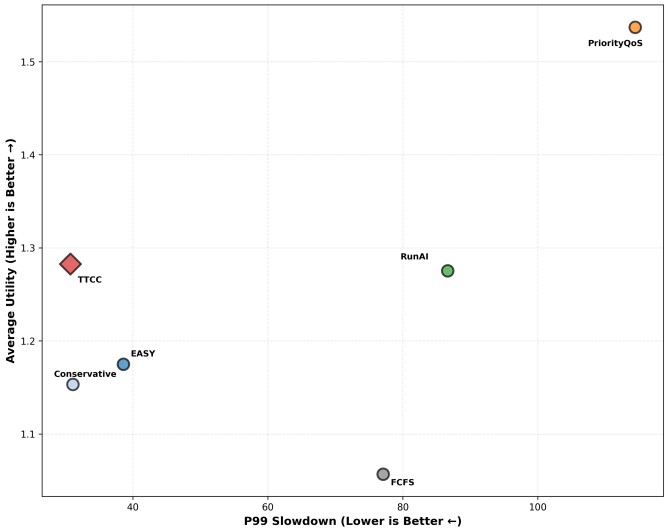

Fig. 4. Mean utility versus mean P99 bounded slowdown across scenarios.

VIII. Discussion

A. Observed Benefits in Practice

TTCC's main benefit is that it turns each completion event into an opportunity for global improvement, even in the waiting-only mode evaluated in this paper. Instead of treating resource releases as purely local queue events, TTCC reconsiders which waiting jobs should run where, using the declared QoS preferences to guide these reallocations. This reduces fragmentation, gives small jobs earlier access to suitable resources, and helps coalesce gang allocations that would otherwise be delayed.

These effects appear consistently across cluster sizes and load regimes: backfilling-level efficiency is preserved, while preference-aware performance improves without introducing the extreme tail behaviour associated with strict priority rules. In practice, this makes TTCC a good fit for multi-tenant research testbeds where operators care about both throughput and perceived fairness, but do not want to rely on aggressive preemption or complex prediction-based policies.

B. Deployment and Applicability

TTCC can operate in *waiting-only* or *migratable* mode depending on checkpointing support. Administrators may cap chain depth, batch completion events, or limit reallocation cadence to balance stability and responsiveness. Because TTCC is invoked on completion events and leaves the underlying scheduler in place, it can be introduced incrementally without re-engineering existing Slurm or Kubernetes deployments.

When to use TTCC. Conservative backfilling remains ideal when tight tail bounds are the primary objective. TTCC is more attractive when testbed operators care about overall user satisfaction and responsiveness under high load, especially in non-monetary environments with heterogeneous QoS needs. In settings where top-tier tenants must be visibly favoured, Priority/QoS may still be preferred despite its longer tails; TTCC offers a middle ground between such rigid priority schemes and purely local heuristics.

C. Limits and Robustness

TTCC assumes truthful QoS and preference reporting, reasonable migration overheads, and single-resource (GPU) semantics. Extending it to composite bundles or designing incentive-compatible mechanisms are natural directions for future work. While the $O(n^2)$ per-event complexity is tractable for testbeds with up to hundreds of jobs, larger deployments may batch events or approximate chain search.

Finally, our evaluation is based on synthetic workloads calibrated to research testbeds. We mitigate variance via common random numbers and a large number of replications, but full validation on production traces and real deployments remains an important next step.

TABLE II
AGGREGATE RESULTS ACROSS CLUSTERS (SMALL/MEDIUM/LARGE) AND LOADS (LIGHT/MEDIUM/HEAVY). VALUES ARE MEANS WITH 95% CIS; P95/P99 ARE MEDIANS WITH 95% CIS.

Sched.	Util.	$\bar{U}$	P95	P99
FCFS	0.761 (0.678–0.844)	1.057 (0.678–1.436)	39 [12–66]	77 [25–129]
EASY	**0.777** (0.687–0.868)	1.175 (0.828–1.522)	19 [7–30]	39 [15–62]
Cons.	0.770 (0.683–0.857)	1.153 (0.806–1.501)	16 [6–26]	**31** [13–49]
PriorityQoS	0.766 (0.680–0.851)	**1.537** (1.347–1.727)	34 [10–59]	114 [30–199]
RunAI	0.764 (0.680–0.849)	1.275 (0.988–1.563)	37 [10–64]	87 [28–145]
TTCC	0.765 (0.681–0.849)	1.283 (0.988–1.578)	**12** [5–19]	**31** [13–48]

IX. CONCLUSION

We bring matching-theoretic reallocation to 6G testbeds by adapting *Top Trading Cycles and Chains* to an event-driven scheduling setting. In our design, each job completion turns the freed resource into an altruist that can trigger preference-improving cycles and chains among queued jobs, while leaving running jobs pinned in the waiting-only variant evaluated here. This yields a global, per-event re-optimization step that is Pareto-efficient under declared preferences and can be layered on top of existing batch or cluster schedulers.

Our simulation study across three cluster scales and multiple load regimes indicates that TTCC can improve preference-aware outcomes over common non-priority baselines without sacrificing the efficiency or responsiveness expected in multi-tenant research testbeds. This makes TTCC a practical option for non-monetary, multi-tenant environments where QoS classes and user-perceived fairness matter alongside utilization.

Future work includes production-grade integration as a Kubernetes extender or Slurm plugin, validation on real testbed traces, and extensions to multi-resource bundles and cost-aware chains with explicit migration budgets. More broadly, we see TTCC as a step toward bringing matching-theoretic guarantees into everyday testbed operations, where heterogeneous resources and heterogeneous needs must be reconciled without relying on pricing.

REFERENCES

[1] D. A. Lifka, "The ANL/IBM SP scheduling system," in *Job Scheduling Strategies for Parallel Processing (JSSPP)*, ser. Lecture Notes in Computer Science, vol. 949. Springer, 1995, pp. 295–303.

[2] J. Skovira, W. Chan, H. Zhou, and D. A. Lifka, "The EASY—loadleveler API project," in *Job Scheduling Strategies for Parallel Processing (JSSPP)*, ser. Lecture Notes in Computer Science, vol. 1162. Springer, 1996, pp. 41–47.

[3] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 6, pp. 529–543, 2001.

[4] L. S. Shapley and H. Scarf, "On cores and indivisibility," *Journal of Mathematical Economics*, vol. 1, no. 1, pp. 23–37, 1974.

[5] A. E. Roth, T. Sönmez, and M. Ü. Ünver, "Kidney exchange," *The Quarterly Journal of Economics*, vol. 119, no. 2, pp. 457–488, 2004.

[6] ——, "Pairwise kidney exchange," *Journal of Economic Theory*, vol. 125, no. 2, pp. 151–188, 2005.

[7] D. Stavropoulos, V. Miliotis, T. Korakis, and L. Tassiulas, "Matching theory application for efficient allocation of indivisible testbed resources," in *2020 IEEE International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt)*, 2020, pp. 1–8.

[8] Kubernetes Authors, "Kubernetes quality of service classes and pod priority/preemption," https://kubernetes.io/docs/concepts/workloads/pods/pod-qos/, 2024, accessed: Oct. 2025.

[9] Y. Yu, Z. Chen, J. Zhang *et al.*, "Volcano: A batch scheduler for container-oriented high-performance workloads," in *Proc. IEEE International Conference on Cloud Computing (CLOUD)*, 2020, pp. 103–112.

[10] Kubernetes SIG Scheduling, "Kueue: Kubernetes-native job queueing system," https://kueue.sigs.k8s.io/, 2024, accessed: Oct. 2025.

[11] A. Floratou *et al.*, "Domino: Resource isolation for shared deep learning clusters," *Proc. ACM Symposium on Cloud Computing (SoCC)*, pp. 1–16, 2022.

[12] S. Fischetti and M. Monaci, "Using constraint programming in scheduling: Models, techniques, and applications," *Annals of Operations Research*, vol. 275, no. 1, pp. 85–110, 2019.

[13] M. Mao, J. Li, and M. Humphrey, "Deeprein: Scheduling deep learning jobs via deep reinforcement learning," in *Proc. IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 240–250.

[14] L. Wang and D. G. Feitelson, "Scheduling parallel jobs using runtime predictions: Theory and practice," *Journal of Parallel and Distributed Computing*, vol. 68, no. 9, pp. 1171–1184, 2008.

[15] M. Snir, R. W. Wisniewski, J. P. Morrison *et al.*, "Addressing failures in exascale computing," in *Proc. ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2014, pp. 1–11.

[16] R. Kettimuthu *et al.*, "Checkpoint/restart and preemption for deep learning training," *Concurrency and Computation: Practice and Experience*, vol. 35, no. 10, p. e7480, 2023.

[17] X. Chen, Y. Zhu, Z. Meng *et al.*, "Preemptible deep learning jobs: Scheduling and checkpointing in the cloud," in *Proc. USENIX Annual Technical Conference (USENIX ATC)*, 2020, pp. 365–378.

[18] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[19] W. Xiao, Y. Bao, M. Musuvathi *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[20] J. Gu, M. Chowdhury, K. G. Shin *et al.*, "Tiresias: A GPU cluster manager for distributed deep learning," in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[21] I. Syrigos, N. Angelopoulos, and T. Korakis, "Optimization of execution for machine learning applications in the computing continuum," in *2022 IEEE Conference on Standards for Communications and Networking (CSCN)*, 2022, pp. 118–123.

[22] I. Syrigos, D. Kefalas, N. Makris, and T. Korakis, "Eelas: Energy efficient and latency aware scheduling of cloud-native ML workloads," in *2023 15th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, 2023, pp. 819–824.