

LFS: Learning-Based Flexible CPU Scheduler

Yeonju Ro
UT Austin

Vincent Zhang
UT Austin

Venkat Arun
UT Austin

Daehyeok Kim
UT Austin

Jiaxin Lin
Cornell University

Christopher J. Rossbach
UT Austin

Zhangyang Wang
UT Austin

Aditya Akella
UT Austin

Abstract

CPU schedulers increasingly serve heterogeneous workloads with distinct latency, fairness, and resource objectives, yet existing schedulers expose fixed heuristics or low-level mechanisms rather than a way to state the objective itself. Operators must therefore translate each goal into scheduling logic by hand, a process that is costly to repeat and offers no guarantees once workloads shift. We present Learning-Based Flexible CPU Scheduler (LFS), a framework that synthesizes CPU scheduling policies directly from a user-specified objective. LFS formulates scheduling as a two-player game between the scheduler and an adversarial environment. Because decision-relevant quantities such as remaining compute and I/O are not observable from the kernel, LFS learns them as latent state with explicit error bounds, within which the adversary may choose any outcome; solving the game yields policies that are optimal within the induced model and robust to prediction error and contention. To make this practical, LFS runs inference asynchronously in `sched_ext`, off the dispatch path, and falls back to more conservative policies when runtime error exceeds the synthesis bound. Across five objectives and six workloads, LFS matches or exceeds the best heuristic baselines, achieving up to $2.5\times$ fewer SLO violations, up to 20% lower P99 latency, and 3–8% lower mean JCT.

1 Introduction

Modern computing infrastructure increasingly consolidates heterogeneous applications with different latency targets, fairness requirements, and resource demands on shared machines. Workload mixes also evolve quickly, driven by services such as Retrieval-Augmented Generation (RAG) pipelines, interactive cloud applications, and emerging edge workloads [12, 28, 31, 43, 46, 65]. In these settings, CPU scheduling directly affects tail latency, slowdown, and isolation. Yet scheduler behavior in practice is still largely determined by fixed heuristics tuned for particular objectives and workload regimes. When the workload mix or the operator’s objective

changes, these heuristics can become misaligned with the desired behavior, leading to avoidable performance loss and, for latency-sensitive services, severe SLO violations [62].

The central challenge is policy derivation: given a target objective, the system must determine a concrete scheduling logic that realizes it under workload variability, partial observability, and shared-resource interference. Existing schedulers expose either a built-in policy with a small set of tuning knobs or a low-level substrate for implementing a custom scheduler. Neither interface directly represents the scheduling objective itself. Instead, operators and OS developers must translate goals such as fairness, latency, or composite service objectives into priorities, tie-breaking rules, and handcrafted heuristics. That translation is workload-dependent, difficult to validate, and costly to redo as workload mixes evolve.

Providing guarantees adds another layer of difficulty. A scheduler that performs well under one workload model may degrade sharply once arrivals, service times, or resource interactions differ from the assumptions under which it was designed. This problem is acute for modern workloads, which are bursty, heterogeneous, and often previously unseen, yet still require predictable performance. Robust policy design therefore requires reasoning under uncertainty in future work, shared-resource interference, and the decision-relevant state exposed to the scheduler. Each of these is difficult to characterize faithfully: interference is complex and often non-composable, OS-visible signals are partial, and workload shifts can invalidate logic tuned to earlier operating regimes.

This paper presents Learning-Based Flexible CPU Scheduler (LFS), a framework that automates CPU scheduling policy design from a user-specified objective. LFS separates *what* to optimize from *how* to schedule: the policy developer specifies the objective as a payoff function over a structured feature vocabulary, and LFS synthesizes the scheduling logic automatically. It does so by formulating scheduling as a two-player game between the scheduler and an adversarial environment, and solving for a policy that is robust to uncertainty in system state and contention.

A central obstacle is that several quantities needed for good

scheduling decisions—remaining compute, remaining I/O, and resource sensitivity—are not directly observable from the kernel. LFS learns these as compact latent state from observable signals and carries explicit uncertainty bounds on the predictions into the game: the adversary may realize any outcome consistent with those bounds. Learned state removes much of the manual effort of constructing explicit performance models, while the game formulation keeps synthesis robust to prediction error and contention. The result is a framework that turns an objective plus uncertainty assumptions into deployable policies with well-scoped optimality guarantees.

Making this approach practical requires system support. Inference is orders of magnitude too slow for the scheduler’s critical path, and switching policies as prediction error changes must not destabilize dispatch. LFS therefore deploys learned state asynchronously in `sched_ext`: models are trained offline, compiled into native code, executed off the dispatch path, and connected to the scheduler through a shared-memory data plane with constant-time lookups. LFS monitors prediction error online and switches to conservative policies when runtime error exceeds the regime assumed during synthesis.

Although LFS is instantiated for CPU scheduling, the underlying approach applies more broadly to systems scheduling and resource management. Many systems policies share the same structure: an operator has a high-level objective, the system exposes only partial and noisy information, and the policy must act under uncertainty about future work and resource interactions. This arises in settings such as I/O scheduling, cluster scheduling, accelerator scheduling, and RPC prioritization, where handcrafted heuristics often compensate for weak objective abstractions and incomplete models. CPU scheduling provides a concrete and consequential setting in which to develop the framework, but the underlying idea – objective-driven policy synthesis over learned state with explicit uncertainty – is more general.

We evaluate policies synthesized under 5 objectives across 6 workloads with varying co-locations. LFS consistently matches or exceeds the best heuristic baseline for each objective, achieving up to $2.5\times$ lower SLO violations, up to 20% lower P99 latency, and 3–8% lower mean JCT, while keeping per-callback data-collection overhead below $1\mu\text{s}$. Inspecting the synthesized policies also reveals that they are not arbitrary: the objective structure determines which signals a policy prioritizes, and prediction accuracy determines how much it relies on estimates of future work—for example, as error grows, an SLO-minimizing policy shifts from a SRTF variant toward plain EDF.

2 Limitations of Existing CPU Schedulers

Modern CPU schedulers offer either a fixed built-in policy with a handful of tuning knobs or a low-level substrate for implementing a custom policy. Neither is a satisfactory interface for today’s settings, where workload mixes are heteroge-

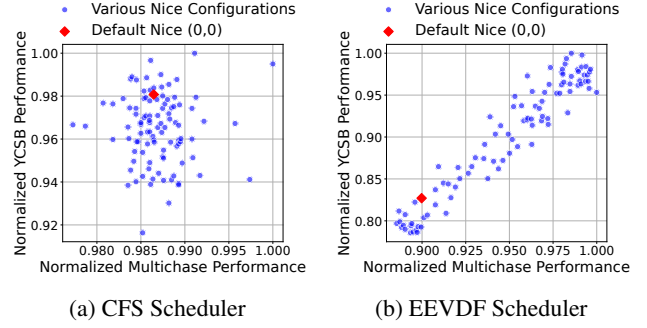


Figure 1: Impact of choosing various `nice` values on maximizing per-workload performance. Performance is normalized to the best configuration; top right is ideal.

neous, objectives are often multi-dimensional, and operating conditions evolve over time. This section shows why: current interfaces make it hard to express scheduling goals directly (§2.1), hard to obtain policies with guarantees (§2.2), and hard to model the system interactions they depend on (§2.3).

2.1 Inflexible Scheduler Interfaces

Consider a cluster operator who wants to enforce fairness across applications co-located on a machine. That goal may be expressed using standard metrics such as Proportional Progress (PP) [19], Average Normalized Turnaround Time (ANTT), or Weighted Speedup [16]. However, commodity CPU schedulers do not let the operator specify any of these objectives directly. Instead, they expose low-level controls such as task priority (`nice`) and scheduling granularity (`sched_min_granularity_ns`), whose effect on the desired objective is indirect, nonlinear, and workload-dependent.

Recent frameworks such as `sched_ext` [51] and `ghOst` [27] make it easier to deploy custom schedulers, but they do not solve this abstraction problem. They lower the cost of implementation, not the cost of policy design. A user who wants to optimize a new objective must still reason from the objective down to a concrete scheduling rule, profile workload behavior, and iteratively tune and validate the resulting policy. In other words, the burden of translating intent into mechanism still falls entirely on the user.

To make this gap concrete, we run a latency-sensitive pointer-chasing workload (`multichase`) alongside a best-effort Redis-backed YCSB workload on a single-core system using Linux CFS and EEVDF. Our goal is to achieve fairness by equalizing slowdown across jobs, where slowdown is defined as completion time under contention normalized by standalone completion time, following prior notions of normalized progress [16, 19]. We ask a simple question: can the exposed scheduler knobs realize this objective in practice? Our results yield three observations.

Observation #1. Default schedulers do not reliably realize the intended objective. Even before tuning, the mismatch

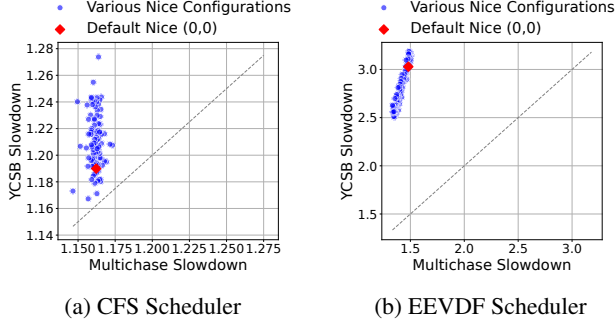


Figure 2: Impact of choosing various `nice` values on achieving fair slowdown (ratio of completion time under contention to completion time in isolation). $y = x$ line is ideal.

between exposed controls and desired behavior is apparent. Under the default configurations, both CFS and EEVDF leave substantial performance on the table, yielding up to 2% and 20% worse per-workload performance than the best configuration we observed (Figures 1a and 1b). More importantly, EEVDF fails to realize the fairness objective itself: the resulting slowdowns remain far from equalized (Figure 2b).

Observation #2. Knob tuning is an unreliable way to implement objectives. Can the objective be approximated by carefully tuning the exposed knobs? We test this by sweeping combinations of 10 possible `nice` values for each workload. For the performance objective, only a small fraction of configurations performs well (Figures 1a and 1b), so finding a good setting already requires a nontrivial search. Worse, the combinatorial search would need to be repeated as workload mixes change. For the fairness objective, tuning is also insufficient: across our sweep, no `nice` configuration achieves fair slowdown (Figures 2a and 2b). Exposed knobs can perturb a fixed policy, but this is not a principled way to realize an arbitrary objective.

Observation #3. Composite objectives are fundamentally harder to realize through fixed knobs. Real deployments rarely optimize a single metric in isolation. In multi-tenant cloud environments, different stakeholders have different goals: application owners want to maximize their own performance, while a cluster operator may impose system-level constraints such as enforcing global fairness across tenants or meeting a cluster-wide power budget. Current schedulers offer no way to reconcile these application- and system-level objectives in a single, cohesive policy. In our experiments, configurations that maximize application performance induce significant unfairness across jobs, while configurations that improve fairness incur substantial performance loss. No single setting of `nice` simultaneously achieves both high performance and fair slowdown, forcing users to prioritize one objective at the expense of others.

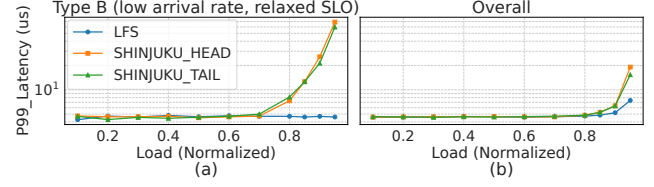


Figure 3: P99 latency of (a) type B requests and (b) all requests under Shinjuku [29] variants, with LFS (§5) shown for reference.

2.2 Regime-Dependent Guarantees

Even when users are willing to implement a custom scheduling policy, ensuring that it satisfies a target objective with meaningful guarantees is difficult. The standard workflow requires analyzing target workloads, constructing a system model that captures how workloads interact with the environment, and designing an algorithm that provably meets the desired objective (e.g., bounded slowdown or SLO compliance). This process is labor-intensive and brittle: it depends on assumptions about workload behavior and system dynamics that are often only approximately true, and those assumptions must be revisited whenever workload characteristics change.

This challenge is especially acute for emerging workloads. Applications such as RAG pipelines and edge services exhibit dynamic, bursty, and previously unseen behavior, yet they remain performance-sensitive and often demand latency or fairness guarantees [20, 52, 61, 62]. In such settings, a scheduler designed around a particular workload model or traffic pattern can perform well in the regime it was tuned for, but fail once the workload shifts outside that regime.

We illustrate this issue using Shinjuku [29], a carefully engineered scheduler designed to reduce SLO violations for latency-critical services. Shinjuku maintains per-type queues and prioritizes the queue whose head request has the highest ratio of current queueing delay to SLO target. We implement two variants in a discrete-event simulator: Shinjuku-head, which reinserts preempted requests at the head of their queue, and Shinjuku-tail, which reinserts them at the tail.

We evaluate Shinjuku under a scenario that reflects common production behavior: asymmetric, bursty arrivals across request types with heterogeneous SLO targets [28, 63], where such burstiness is itself a common cause of SLO violations [60]. We construct a synthetic workload with two request types, A and B, where type A has both a stricter SLO and a substantially higher arrival rate.

Under this workload, Shinjuku’s ratio-based prioritization systematically over-favors type A. The higher arrival rate inflates A’s queueing delay, while the tighter SLO reduces the denominator of the priority ratio, causing A to dominate scheduling decisions even when doing so worsens the overall P99 latency across request types (Figure 3(b)). Type B is therefore persistently deprioritized and suffers disproportionately high tail latency (Figure 3(a)).

This example illustrates a broader limitation of manual

```

1 fn calculate_metrics(&self) -> SchedulingMetrics {
2   let mut sum_penalty = 0.0;
3   for job in &self.jobs {
4     let ratio = job.finish_time / job.deadline;
5     let penalty = if ratio <= 1.0 { 0.0 }
6                   else if ratio <= 3.0 { 1.0 }
7                   else { 2.0 };
8     sum_penalty += job.weight * penalty;
9   }
10   SchedulingMetrics::new([sum_penalty])
11 }

```

Figure 4: Example payoff function that minimizes SLO violations with a stepwise penalty based on violation severity.

scheduler design: the resulting guarantees are inherently regime-dependent. A policy may perform well under the workload patterns it was designed around, yet fail once arrivals, service-time asymmetries, or mixtures of SLO needs change. As workloads become more heterogeneous and dynamic, obtaining guarantees by hand-designing and repeatedly retuning heuristics becomes increasingly brittle.

2.3 Difficulty of Interference Modeling

Even when workload characteristics are known, obtaining scheduling guarantees requires reasoning about how co-running jobs interfere through shared resources such as the LLC and memory bandwidth. A natural approach is to build an explicit system model that predicts how such interference affects latency and throughput. In practice, this is difficult. Cross-workload interference is complex and often non-composable: cache thrashing, bandwidth saturation, and related effects emerge from interactions among multiple tasks and resources, so small changes in co-runner mix or execution phase can cause large performance shifts [5, 21]. OS-level signals further limit fidelity because they expose aggregate symptoms of contention rather than root causes. As a result, even refined interference models [42, 44] remain incomplete. This creates a tension for scheduler design: strong guarantees appear to require precise models, yet the interactions most important to performance are hard to model faithfully. This motivates approaches that synthesize policies under model uncertainty rather than on a fully specified system model.

3 LFS Overview

To address the limitations in §2, we propose LFS, an automated policy synthesis framework for CPU scheduling. Instead of exposing low-level knobs (§2.1), it accepts a high-level objective directly; instead of a policy tuned for one regime (§2.2), it synthesizes a policy that is optimal within an uncertainty-aware model; and instead of requiring an explicit interference model (§2.3), it learns decision-relevant state with explicit error bounds. LFS then deploys the policy into the Linux kernel, decoupling user intent from policy

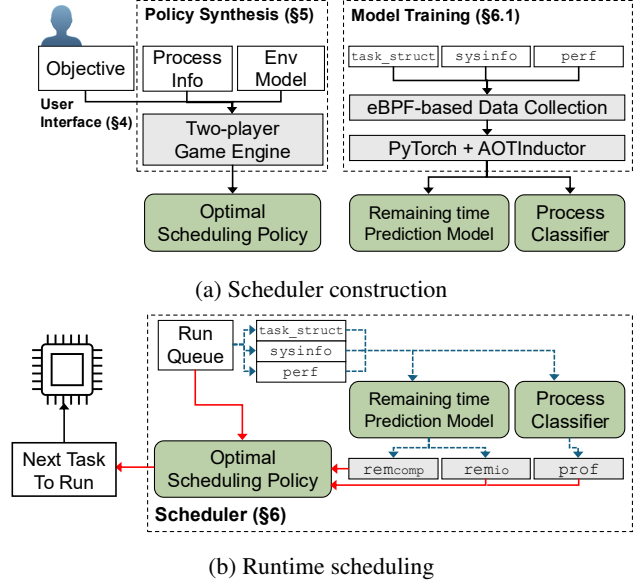


Figure 5: Overview of LFS. (a) Scheduler construction combines policy synthesis from the user-specified objective with model training. (b) At runtime, the policy uses predicted per-task state to select the next runnable task. The blue dashed line (---) denotes the inference path running in the background, and the red solid line (→) represents the scheduler’s critical path. `rem_comp` and `rem_io` are remaining compute and remaining I/O time.

implementation. All heavy lifting—model training, policy synthesis, and scheduler compilation—happens offline, producing a kernel-loadable scheduler before any scheduling decision is made.

User interface. (§4) A policy developer interacts with LFS by writing a payoff function—a short code snippet that scores execution outcomes over a finite horizon using process-level metrics such as slowdown, latency, or fairness. Figure 4 shows one example. LFS then handles telemetry collection, model training, policy synthesis, and deployment.

From objective to policy. Upon invocation, LFS executes three offline pipelines in sequence (Figure 5). First, the model training pipeline (§6.1) collects kernel data structures and system counters via eBPF and trains an ML model that maps raw features to higher-level, decision-relevant quantities for each process, including remaining compute time, remaining I/O time, and resource affinity. Next, the policy synthesis pipeline combines the user-defined payoff function with an environment model—comprising the trained predictor and a contention-delay model (§5.3)—and solves a two-player scheduling game over representative execution scenarios via minimax optimization [47, 50] (§5). The resulting state-action traces are then distilled into a decision-tree policy. Finally, the distilled policy and prediction models are compiled via AOTInductor [3] into native C++ code and linked into the scheduler binary, which is deployed via `sched_ext` [51]

(§6.2).

Online execution. At runtime, the prediction model continuously estimates per-task quantities such as remaining compute time and remaining I/O time, and the distilled policy uses these estimates to rank runnable tasks at each scheduling decision point. To avoid placing inference on the scheduler’s critical path, predictions are computed asynchronously in user space and written to a shared memory-mapped array ($\dashrightarrow$). The BPF scheduler retrieves them via constant-time lookups ($\rightarrow$), keeping model execution off the dispatch path (§6.3). LFS also monitors online prediction error and, when the observed error exceeds the bound assumed during synthesis, switches to a more conservative policy synthesized for a larger bound, keeping the deployed policy’s assumptions consistent with runtime conditions under workload shifts (§5.8).

4 Policy Design Interface

LFS exposes a policy design interface centered on two elements: a **payoff function** specifying the scheduling objective, and a **feature vocabulary** defining the observable quantities available to the objective and synthesis engine.

4.1 Payoff Function and Feature Vocabulary

The payoff function is a user-supplied code snippet that maps a complete execution trajectory—the sequence of scheduling decisions and resulting job outcomes over a finite horizon—to a score vector. This score is the policy developer’s specification of the objective. By default, score vectors are compared lexicographically under minimization, from the first component to the last. Objectives that require maximization, such as throughput, can be expressed by negating the corresponding component.¹ Users may also define a custom comparator when a different ordering is needed.

The policy developer writes the objective using LFS’s feature vocabulary (Table 1), which provides a structured view of the system state visible to the synthesis engine and the deployed policy. The vocabulary includes (a) global system features shared across jobs, such as clock and runqueue statistics; (b) per-job observables obtained directly from Linux and perf counters or inferred by the prediction model; and (c) derived features, such as normalized quantities and ranks, that make relative notions like urgency, progress, and fairness easier to express.

4.2 Expressiveness

Using the feature vocabulary, payoff functions can express a wide range of scheduling objectives spanning both perfor-

¹The payoff function is evaluated once at the end of the horizon, over the entire trajectory, and the resulting value is propagated back through the game tree during minimax optimization.

Category	Features
System-level	current_time, num_jobs_in_queue
Per-job observables	obs_burst_rem, is_compute_burst, svc_time, qdelay, progress, deadline, io_rem, in_io, runnable, finish_time
Derived / Relative	slack, slack_ratio, rank_slack, rank_progress, rank_qdelay, rank_obs_burst_rem, rel_progress, rel_qdelay, is_most_urgent, is_most_progressed, norm_obs_burst_rem

Table 1: Feature vocabulary exposed to the payoff function and policy synthesis engine.

mance and fairness.

Performance objectives. Objectives targeting latency, SLO compliance, completion time, and throughput can be expressed using penalties over features such as `deadline` and `finish_time` at job completion (Figure 4). These penalties may be binary (deadline miss), tiered, or continuous (e.g., proportional to tardiness). Aggregate metrics such as average or weighted completion time and makespan can likewise be computed (Figure 18 in §A.2), supporting both tail-sensitive and throughput-oriented objectives.

Fairness objectives. Fairness objectives can be formulated in terms of either job progress or allocated service. Progress-based notions, expressed using features such as `progress` and `rel_progress`, capture how evenly jobs advance toward completion. Service-based notions, expressed using `svc_time`, capture how CPU time is distributed across jobs, optionally with weights reflecting job importance. These formulations support common objectives such as max-min fairness, proportional fairness, and deviation-based fairness (Figures 19 to 21 in §A.3–A.5).

Multi-objectives and system constraints. LFS supports composite objectives in two forms. First, users may combine multiple metrics into a single scalar payoff, for example by adding a fairness penalty to a performance objective. This supports soft constraints and weighted tradeoffs, such as optimizing performance while penalizing excessive unfairness (Figure 17 in §A.1). Second, users may specify a vector-valued payoff together with an ordering over its components, enabling priority-ordered objectives such as minimizing tail latency first and maximizing throughput second. In both cases, the developer must specify the desired tradeoff or priority explicitly; LFS does not infer it automatically.

These constructs cover a broad range of CPU scheduling objectives, but not every possible policy goal; objectives that require additional observability or hard per-step constraints fall outside the current interface (§A.6).

5 Automated Policy Design

Given an objective, LFS synthesizes a scheduling policy by formulating scheduling as a two-player game between the scheduler and an adversarial environment and solving the

game via minimax optimization. The central challenge is that several decision-relevant variables are not directly observable and must be inferred from noisy signals. This introduces uncertainty into the scheduler’s view of system state: the policy must remain robust to the worst-case realizations consistent with prediction error. LFS addresses this by combining learned latent state with explicit uncertainty bounds and solving for a policy that is robust within the resulting game model. This section presents the game (§5.1), the learned state and contention models that define the adversary’s uncertainty (§5.2–5.3), the formal game definition and policy distillation (§5.4–5.5), the scope of the resulting guarantees and the gap introduced by approximation (§5.6–5.7), and a guardrail for when runtime error exceeds the assumed bound (§5.8).

5.1 CPU Scheduling Game

Each game scenario captures the execution of a set of jobs over a bounded horizon. The game runs in a lightweight simulator of CPU scheduling dynamics that models task execution, arrivals, and preemption.²

Two players alternate turns. At each turn, the scheduler (the *player*) selects the next runnable job to execute so as to optimize the user-specified objective—latency, makespan, fairness, or any other metric expressible through the payoff function. The adversarial environment (the *adversary*) then resolves uncertain aspects of execution in the way that is worst for that objective: it selects realizations of quantities such as burst lengths, inter-arrival delays, and contention-induced delay, subject to the bounds defined by the uncertainty model. Execution then proceeds until the next scheduling event. Solving against this adversary yields decisions that are robust to every realization the model permits.

The game terminates when either a fixed number of scheduler decisions have been taken (i.e., a bounded horizon h) or all jobs complete. Solving the game yields a set of state-action traces that characterize optimal decisions within this formulation for the specified objective. LFS collects such traces across many scenarios and distills them into a decision-tree policy, which serves as the deployable scheduler. The scope of the resulting optimality guarantee is discussed in §5.6.

5.2 Learned State and Error Bounds

Both players act on a representation of the current system and job state. From the scheduler’s perspective, however, the system is partially observable: the kernel exposes queue state and system counters, but not several decision-relevant variables needed to optimize user-defined objectives, such as remaining compute, remaining I/O, or contention sensitivity. While such quantities could in principle be estimated using carefully engineered analytical models [42, 44], constructing an accurate

²LFS supports both preemptible and non-preemptible settings; unless otherwise noted, evaluations use the non-preemptible configuration.

Scope	Source	Example Features
Per-process	task_struct	sum execution times, total NUMA faults, blk/swap delay, ...
Per-core	perf	cache references, dTLB/iTLB misses
System-wide	sysinfo	loads, RAM and swap (e.g., total/free/shared/buffered)

Table 2: Input features from Linux used in model training.

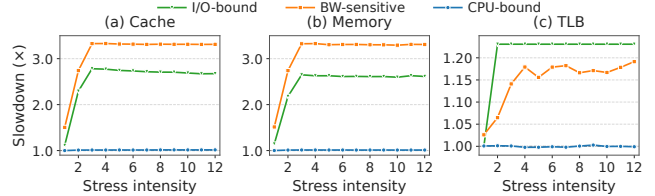


Figure 6: Slowdown per stress type and intensity. Stress intensity is quantified by per-core LLC miss rate (cache), memory bandwidth utilization (memory), and dTLB miss rate (TLB).

model is difficult in practice because system interactions are complex and nonlinear (§2.3).

To address this limitation, LFS uses an ML model (§6.1) that takes observable signals (Table 2) as input and infers latent, decision-relevant state variables, including remaining compute, remaining I/O, and resource sensitivity. These quantities are not directly observable from the kernel, but they compactly summarize process state in a form that is predictive of future execution. Rather than requiring a fully explicit analytical model of the environment, LFS makes scheduling decisions over this learned state together with the uncertainty model described below.

Each inferred variable is accompanied by an explicit error bound derived from the model’s prediction error. These bounds quantify uncertainty in the latent state and define the adversary’s action space. Thus, although the scheduler never observes the true underlying state, policy synthesis accounts for every realization consistent with the error bounds.

5.3 Contention Delay Bounds

One latent variable inferred by LFS is a process resource profile P_i , which summarizes how a process interacts with shared resources such as the LLC and memory bandwidth. When co-scheduled processes place pressure on the same shared resource, they can interfere with one another, increasing execution time through cache thrashing, bandwidth contention, and related effects [18, 36, 42, 44]. Rather than predicting exact interference online, LFS models contention as additional execution delay, bounded by a function of the process’s resource profile and the current system state.

Workload-aware Delay Bound. At each scheduling step, the adversary may impose an additional execution delay $d_i^t \in [0, \mathcal{D}_{\max}^t]$, where the bound $\mathcal{D}_{\max}^t = f(P_i^t, S^t)$ depends on the

process's predicted resource profile P_i^t and the current system state S^t . Here, $\mathcal{D}_{\max}^t$ denotes the maximum additional wall-clock time that process i may incur from contention during a single scheduling interval.

We obtain f empirically through stress experiments that induce contention on specific shared resources and measure worst-case slowdown relative to isolated execution over fixed intervals. These measurements yield a calibrated mapping from resource profile and system condition to a per-interval delay bound. Figure 6 shows representative slowdown curves used to derive these bounds.

We assume that the cluster operator specifies an operating regime—through operational policies such as admission control or target utilization—that bounds the contention applications can experience;³ the measured f then gives the corresponding worst-case slowdown. These bounds cap interference under the operator's policy and constrain the adversary in the game, so that it models worst-case interference while remaining grounded in the actual operating conditions.

5.4 Formal Game Definition

We now formalize the scheduling game that translates the user-specified payoff function into a robust policy. The scheduler acts on estimates provided by the prediction model, while the adversary resolves uncertain quantities within the bounds of the uncertainty model. LFS thus optimizes not for a single predicted future but for the worst case those bounds permit.

State Definition. Each job i is represented as an alternating sequence of compute and I/O bursts, $B_i = (b_{i,1}, b_{i,2}, \dots, b_{i,M_i})$, where each burst $b_{i,k} = (kind_{i,k}, \ell_{i,k})$ specifies whether the burst is compute or I/O and its length. At scheduling step t , the system state is $X_t = \{(\hat{B}_i^t, k_i^t, \hat{r}_i^t, P_i^t, in_io_i^t) \mid i \in \{1, \dots, N\}\}$, where k_i^t is the index of job i 's current burst, $\hat{r}_i^t$ is the estimated remaining length of that burst, P_i^t is a resource profile, and $in_io_i^t$ indicates whether job i is currently in I/O. The estimate $\hat{r}_i^t$ is produced by the prediction model (§6.1) and may differ from the ground truth by error margins ϵ_c and ϵ_{io} for compute and I/O, respectively. The error bounds ϵ_c and ϵ_{io} are independently configurable. We use ϵ generically for the applicable bound. When a policy is indexed by a single ϵ , it uses $\epsilon_c = \epsilon_{io} = \epsilon$, as in all our evaluations.

Scheduler's Action. At scheduler turn t , the scheduler selects a job i to dispatch:

$$a_t \in \left\{ i \mid \neg done_i^t \wedge \neg in_io_i^t \wedge kind(b_{i,k_i^t}) = \text{COMPUTE} \right\}.$$

That is, a job is runnable if it has not finished, is not in I/O, and its current burst is a compute burst. If no job is runnable, time advances implicitly until one becomes available.

³For example, if admission control limits the number of runnable jobs per core or caps memory-intensive jobs, these constraints bound contention.

Adversary's Action. After the scheduler selects a job j to dispatch at turn t , the adversary chooses three quantities. First, it selects the realized compute progress of job j during the current scheduling interval: $\hat{w}_j^t \in [\hat{r}_j^t(1 - \epsilon_c), \hat{r}_j^t(1 + \epsilon_c)]$. The scheduler observes only the predicted remaining compute $\hat{r}_j^t$, while the adversary instantiates an actual realization within the uncertainty set induced by ϵ_c . The adversary thus does not alter the workload arbitrarily; it chooses only among outcomes consistent with the prediction-error bound.

Second, the adversary chooses a contention delay $d_j^t \in [0, \mathcal{D}_{\max}^t]$ for the scheduled job, representing additional wall-clock time incurred due to interference on shared resources (§5.3). The total elapsed wall-clock time of the interval is therefore $w_j^t = \hat{w}_j^t + d_j^t$.

Third, the adversary determines I/O progress for all jobs with outstanding or pending I/O as $\hat{v}_k^t \in [\max\{0, \hat{r}_k^t(1 - \epsilon_{io})\}, \hat{r}_k^t(1 + \epsilon_{io})]$. For each job $k \neq j$, let

$$p_k^t = \begin{cases} \min(\hat{v}_k^t, w_j^t), & in_io_k^t \vee kind(b_{k,k_i^t}) = \text{I/O}, \\ 0, & \text{otherwise.} \end{cases}$$

This determines when background jobs complete I/O and become runnable again, allowing the game to model unfavorable task-arrival patterns during execution. The adversary's action is thus $(\hat{w}_j^t, d_j^t, \{p_k^t\}_{k \neq j})$.⁴

State Transition and Termination. After the scheduler dispatches job j and the adversary selects its action, the state updates as follows. For the scheduled job, remaining compute decreases by realized progress: $\hat{r}_j^{t+1} = \max(0, \hat{r}_j^t - \hat{w}_j^t)$. The contention delay d_j^t advances wall-clock time but is not useful compute progress, so it does not reduce the burst's remaining compute. For each non-scheduled job $k \neq j$ with an active I/O burst, remaining I/O decreases by its realized progress: $\hat{r}_k^{t+1} = \max(0, \hat{r}_k^t - p_k^t)$. The global clock advances by the full wall-clock duration: $T^{t+1} = T^t + w_j^t$. When a burst completes ($\hat{r}_i^{t+1} = 0$), the job advances to its next burst, reinitializing the remaining length and updating the I/O flag accordingly. The game terminates after at most $2h$ plies: h scheduler decisions interleaved with h adversary turns.⁵

Across the sequence of player and adversary actions, we do not assume a particular error distribution or independence across jobs and scheduling steps. Real errors may arise from complex, highly correlated phenomena that violate such assumptions and would invalidate guarantees built on them. LFS therefore makes no such assumptions: the adversary may choose any sequence permitted by the error bounds and the

⁴Motivated by practical scheduling granularity [34], we define a minimum positive progress increment for each compute or I/O burst.

⁵Evaluating payoffs is complicated when jobs remain unfinished at the end of the horizon. To address this, we introduce a warmup phase before the game; across multiple timesteps, randomly selected jobs make partial progress. At $h = 0$, each job starts with such progress accumulated during warmup; we set the warmup phase long enough such that all jobs complete within the game horizon.

scheduling dynamics, so the resulting decisions are robust within the induced game model.

5.5 Policy Distillation

To deploy a minimax-derived policy in `sched_ext`, we distill the game traces—state-action pairs (X_t, a_t) —into a deployable scheduling policy via imitation learning. The key challenge is that the runqueue is a variable-sized set of runnable jobs, so the deployed policy must act as a permutation-invariant ranking rule rather than a classifier over a fixed-length input. We therefore represent the state as a set of per-job feature vectors rather than one fixed-length vector.

For each trace step, we construct one training instance per runnable job, using its feature vector as input and labeling the minimax-selected job as positive and the others as negative. We then train a per-job scoring model in the form of a decision tree so that the minimax-selected job receives the highest score among runnable jobs. At inference time, the learned model scores all runnable jobs independently and dispatches the argmax. This formulation is permutation-invariant, naturally supports variable-sized runqueues, and yields a compact ranking policy that captures job-level decision structure rather than memorizing job identities. This synthesis provides a systematic way to discover new scheduling policies, uncovering decision structures that differ from established heuristics which we will discuss in §7.2.

5.6 Scope of Optimality

The synthesized policy is optimal within the induced scheduling game: no alternative policy operating under the same state representation and uncertainty model can achieve a better worst-case payoff for the specified objective. This guarantee is relative to the model; it does not formally imply superiority over heuristics such as CFS, EEVDF, or round-robin, which operate outside it (e.g., without learned state). We therefore compare against such baselines empirically (§7.1).

The error bound ϵ parameterizes a family of policies with different robustness and performance profiles. The optimal policy derived under an oracle model ($\epsilon = 0$), denoted `minimax-0`, represents the best achievable performance when predictions are perfect. Incorporating bounded prediction error ($\epsilon > 0$) yields `minimax- $\epsilon$` , which is optimal against a stronger adversary that can exploit prediction inaccuracy within the ϵ bound. As ϵ increases, the adversary gains more freedom and the guaranteed payoff degrades, but the policy remains the best possible within that uncertainty class. Thus, LFS guarantees that either the induced game is true, and hence those guarantees hold or the predictions are incorrect, which is easily detectable and repairable with guardrails.

5.7 Bounded-Depth Approximation

`minimax- $\epsilon$` corresponds to the full-horizon formulation: the game proceeds from the initial state until all jobs complete, and the scheduler optimizes a global objective over the full execution under worst-case dynamics. Solving this game exactly is computationally intractable. We therefore approximate it with a bounded-depth search over a d -step lookahead, denoted `alg-d- $\epsilon$` . By construction, `alg-d- $\epsilon$` is only *locally* optimal—optimal with respect to the truncated lookahead—and may deviate from the global optimum of `minimax- $\epsilon$` ; as $d \rightarrow \infty$, the two converge. We empirically evaluate the effect of search depth on policy quality in §7.4–7.5 and find that the approximation gap is negligible in practice.

5.8 Error-Bound Guardrail

The guarantees of LFS rely on the predictor satisfying the error bound ϵ assumed during policy synthesis. Under workload distribution shift, this assumption may be violated, and the guarantee of the policy synthesized for that bound no longer applies. LFS therefore employs a guardrail that monitors prediction error online and switches to a more conservative policy when needed. The scheduler calculates error as the mean absolute percentage error (MAPE) between predicted and observed burst lengths over a sliding window of the most recent W decisions (we use $W = 500$). When the rolling MAPE exceeds the assumed ϵ , the scheduler transitions to a policy synthesized with a larger error bound, or to a fallback policy if the observed error exceeds the maximum supported error bound (e.g., $\text{MAPE} > 50\%$). This restores consistency between the deployed policy and the uncertainty level observed at runtime. When the rolling MAPE falls back below the original bound ϵ , the scheduler reverts to the original policy; to avoid oscillation, it requires the error to remain below the bound for K consecutive windows before reverting (we use $K = 5$).⁶

6 Learned Scheduler Deployment

This section describes the systems mechanisms that make learned-state scheduling practical: data collection and training, model packaging, asynchronous inference over a shared-memory data plane, and policy hot-swapping.

6.1 Training Predictive Models

Training data. We train models to infer the latent state variables introduced in §5.2, including remaining compute and remaining I/O, from kernel-observable signals. Inputs combine per-task fields from `task_struct` with system-level counters from `perf` and `sysinfo` (Table 2). We collect training traces on the target platform using an eBPF-based telemetry

⁶System support for policy hot-swapping is discussed in §6.4.

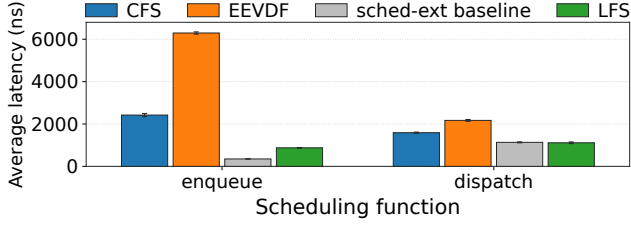


Figure 7: Scheduling handlers overhead. For CFS/EEVDF, we measured runtime of `enqueue_task_fair()` for enqueue and `pick_next_task()` for dispatch.

pipeline that shares the same observability substrate as the deployed scheduler. During collection, inference is disabled and per-event traces are exported for offline preprocessing and labeling.⁷ Labels for remaining compute and remaining I/O are derived offline from execution traces, and model training is performed entirely in user space.

Model architecture. CPU scheduling is inherently temporal: future behavior depends not only on the current snapshot but also on recent execution history. We therefore evaluate sequence models—LSTM, Transformer, and state space models (SSMs)—rather than relying only on the snapshot-based MLPs common in prior ML-for-systems work [10, 25]. Under a fixed model-size and training budget, the Transformer provides the best accuracy–latency tradeoff among sequence models (§7.7), so we use it in the final system.⁸

6.2 Model Compilation and Loading

For flexible model development, we implement prediction models in PyTorch. To deploy them in a non-Python scheduling environment, we use `torch.export` with AOTInductor [3] to compile all artifacts into a single package, which the `sched_ext` userspace scheduler loads to run inference directly in C++, eliminating interprocess communication with a Python runtime. This design consolidates the scheduler logic into a single userspace process, retaining Python’s development convenience while reducing inference latency relative to native Python (§7.7).

6.3 Asynchronous Inference

ML inference is fundamentally incompatible with the scheduling critical path: scheduling handlers run in hundreds to thousands of nanoseconds (Figure 7), whereas even a modest 3-layer MLP with hidden size 128 takes 1–2.8 ms per forward pass—a gap of three to four orders of magnitude. We therefore run inference asynchronously in a background thread. The scheduler reads the latest predictions at decision time, so model execution never blocks scheduling operations. To

⁷Although the data plane is shared and CSV export overhead is minimal, inference is disabled during collection, so data collection runs on a separate machine rather than on the deployed scheduler.

⁸Training details are in §C.

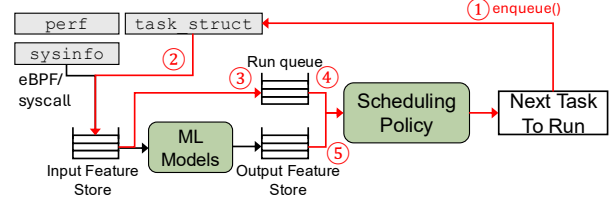


Figure 8: Data flow in LFS. The red arrow (→) indicates the scheduler’s critical path.

enable this decoupling, we introduce input and output feature stores that hold model inputs and predictions, respectively.

Scheduler’s Critical Path. Figure 8 shows the data path of the scheduling pipeline. When a task yields the CPU, either by preemption or completion (①), its latest `task_struct`-derived features are written to the input feature store (②) and the task re-enters the runqueue (③). The prediction model runs asynchronously over this shared input state and writes updated outputs to the output feature store. At dispatch time, the scheduler reads the runqueue (④) and the latest per-task predictions via constant-time lookups (⑤), so inference remains entirely off the dispatch path.

Kernel-Userspace Data Transfer. ② and ⑤ involve kernel-user space data transfer that may cause *timeliness* problems: prediction should reflect recent process state, or stale values will mislead scheduling decisions. Timeliness is determined by how quickly kernel-observable signals flow to the model (②) and how quickly predictions flow back to the scheduler (⑤). Delayed updates increase prediction error and degrade scheduling performance. Moreover, the policy’s robustness guarantee applies only if the predictions used at dispatch satisfy the error bound assumed during synthesis. The data plane must therefore provide low-latency access while avoiding excessive copying and syscall overhead.

To meet these requirements, we design a shared-memory data plane using memory-mapped `BPF_MAP_TYPE_ARRAY` structures for both feature export and prediction import. Compared to alternatives such as ring buffers and hash maps, array-based storage provides predictable $O(1)$ access, avoids hashing and pointer indirection, and enables zero-copy sharing via `mmap`; we evaluate these trade-offs in §7.6.

Array Reindexing. A key challenge in array-based designs is indexing. A naive approach would index entries by `pid`, but this is unsuitable: the PID space is large and sparse, leading to poor memory efficiency and increased cache and TLB pressure. Hash maps avoid sparsity but introduce hashing overhead and pointer indirection, which are undesirable on the critical path and incompatible with `mmap`-based sharing.

Instead, we reindex active tasks into a dense identifier space $[0, N)$ and use these identifiers as offsets into the shared arrays. The mapping is maintained out of band: a free slot is assigned when a task becomes active and reclaimed upon exit. This enables compact memory layout and ensures that each lookup

requires a single array access, providing predictable, low-latency performance suitable for scheduler-critical operations.

6.4 Policy Hot-Swap

Delayed or missing prediction updates can leave the scheduler using stale predictions. The guardrail (§5.8) monitors rolling MAPE and switches to a policy synthesized for a larger error bound when the current policy’s bound is exceeded, or to a fallback policy when the error exceeds the maximum supported bound. A key systems challenge is performing policy transitions without disrupting the scheduler’s critical path. Because policies are learned ranking functions, a policy change invalidates the ordering of tasks in the dispatch queue (DSQ). Recomputing all priorities eagerly would introduce prohibitive overhead. Instead, LFS adopts a *lazy transition*: tasks already in the queue retain their previous scores, while newly arriving tasks are evaluated under the new policy. All scores are normalized to $[0, 1]$, avoiding cross-policy scaling inconsistencies. The resulting inconsistency is transient and bounded: it affects only tasks already enqueued at the time of the switch; as tasks dispatch, yield, and re-enter the DSQ, the queue naturally converges to the new policy without a bulk reordering pass. This preserves hot-swap responsiveness while keeping the transition off the critical path.

7 Evaluation

Evaluation Setup. We conduct experiments on Dell EMC PowerEdge C6525 machines (CloudLab c6525-25g nodes) equipped with a 16-core AMD EPYC 7302P CPU, 128 GB of memory, and 25 Gbps networking via Mellanox ConnectX-5 adapters with PCIe Gen 4.0. We compare LFS against CFS (v5.15), EEVDF (v6.14) as general-purpose Linux baselines, EDF as a representative deadline-based scheduler, and Shinjuku [29] as a specialized tail-latency scheduler. Our latency-critical workloads are Redis, Nginx, and DBBench (RocksDB), covering in-memory data services, web serving, and storage; our throughput-oriented workloads are `mcf`, `x264`, and `xz` from SPEC2017 [53], batch workloads for evaluating completion efficiency and fairness. We use application deadlines of 10 ms, 5 ms, and 2.5 ms for DBBench, Redis, and Nginx, respectively [11, 12, 17, 29]. All experiments use 8 logical CPU cores. The end-to-end cost of producing a policy is approximately 10 hours for data collection on the target machine, 4 hours for model training on a single NVIDIA A10 24GB GPU, 3 hours for synthesis, and less than 1 minute for policy distillation.

7.1 Performance Across Objectives

We evaluate whether policies synthesized under worst-case assumptions also achieve competitive empirical performance across different scheduling objectives. Figure 9 compares

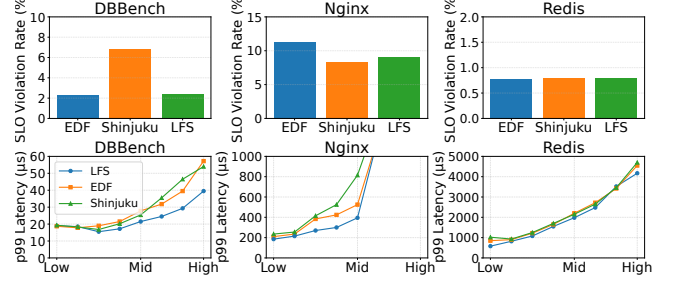


Figure 9: Performance of policies synthesized under objectives that minimize (a) SLO violation rate (top) and (b) P99 latency (bottom), compared with EDF and Shinjuku.

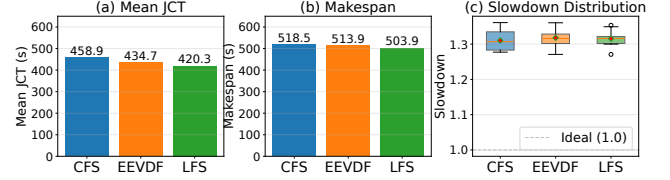


Figure 10: Performance of policies synthesized under objectives that minimize (a) mean JCT, (b) makespan, and (c) progress variance (fairness), compared with CFS and EEVDF. (c) shows the resulting per-job slowdown distribution.

LFS with EDF and Shinjuku; its top row shows the SLO violation rate. The heuristic schedulers are workload-sensitive: EDF is strong on DBBench but degrades on Nginx, while Shinjuku shows the opposite trend. Because Shinjuku prioritizes requests with a high ratio of queueing delay to SLO target (§2.2), our per-application deadlines (10, 5, and 2.5 ms for DBBench, Redis, and Nginx) lead it to favor Nginx, the workload on which it performs best. LFS avoids these failures by synthesizing a policy for the objective itself, matching or outperforming the best baseline on every workload.

The bottom row of Figure 9 evaluates P99 latency across load levels. As load increases, EDF and Shinjuku exhibit sharp latency spikes, whereas LFS maintains a flatter curve. Gains are most pronounced for Nginx and DBBench; Redis performs similarly under all schedulers.

Figure 10 compares LFS policies for system-level objectives—(a) mean JCT, (b) makespan, and (c) fairness—against CFS and EEVDF on the SPEC workloads. LFS outperforms both baselines on mean JCT and makespan (Figure 10(a)–(b)). For fairness (Figure 10(c)), CFS and EEVDF show wide slowdown distributions with long tails, while LFS concentrates slowdown tightly near 1.0, i.e., it enforces more uniform progress across co-located jobs.

Together, these results demonstrate that LFS translates different scheduling objectives into policies with strong empirical performance. Policies synthesized under worst-case assumptions achieve competitive SLO violation rates and tail latency, lower mean JCT and makespan, and more uniform slowdowns across the evaluated workloads.

Variant	Prediction error bound ϵ		
	Reliable (0–0.2)	Moderate (0.2–0.5)	High (≥ 0.5)
<i>Minimize SLO violations (penalty shape)</i>			
Binary	Triage-SRTF*	Triage-LLF*	EDF
Stepwise	LLF	EDF	EDF
Continuous	EDF/SRTF hybrid	EDF	EDF
<i>Optimize performance (target metric)</i>			
Mean JCT	Overlap-aware†	SRIF‡	RR/LAS
Queuing delay	SRIF‡	SRIF‡	RR/LAS
Makespan	LRTF-MPF§	LRTF-MPF§	RR/LAS
<i>Maximize fairness (definition)</i>			
Max-min	LRJF	LRTF-MPF§	RR/LAS
Proportional	SRIF‡	SRIF‡	MPF
Devotional	SRTF-MPF	SRTF-MPF	MPF

* Triage: identifies and deprioritizes jobs that cannot meet their deadlines.

† Overlap-aware: prioritizes jobs with both pending compute and I/O to maximize overlap.

‡ SRIF: Shortest Remaining I/O First.

§ LRTF-MPF: Longest Remaining Time First, ties broken by Minimal Progress First.

Table 3: Policies synthesized with LFS, projected onto the nearest known heuristic, by objective and prediction error bound ϵ (high includes no oracle).

7.2 Structure of Synthesized Policies

We examine how the scheduling objective and prediction uncertainty shape the structure of synthesized policies, as summarized in Table 3. Each synthesized policy is a decision tree that does not exactly match any existing heuristic. To understand how policy structure changes with the objective and uncertainty, we project each one onto the nearest known heuristic from the literature: we implemented over 44 heuristics and manually compared the trees against them.

Our key finding is that synthesized policies are not arbitrary: they organize into a small number of behavioral regimes. First, **prediction accuracy** determines whether the scheduler can safely exploit estimates of future work or must rely on non-predictive signals. Consider the binary SLO-violation objective. We use *triage* as a descriptive label for the learned behavior of deprioritizing jobs predicted to miss their deadlines, directing resources toward jobs whose deadlines remain achievable. When predictions are reliable ($\epsilon = 0\text{--}0.2$), the synthesized policy most closely matches a triage-based variant of SRTF (Triage-SRTF) that deprioritizes jobs with negative laxity.⁹ Because laxity depends on estimates of remaining work, this regime requires high prediction accuracy. As ϵ increases, the policy shifts to Triage-LLF ($\epsilon = 0.2\text{--}0.5$) and then to EDF ($\epsilon \geq 0.5$).¹⁰ This progression gradually drops features that depend on predicted remaining work: Triage-SRTF uses it for both triage and ordering, Triage-LLF only

⁹Laxity is defined as the remaining time margin before a job becomes infeasible, computed as deadline – current time – $C_{\text{remaining}}$ [9, 35].

¹⁰LLF is Least Laxity First; Triage-LLF additionally skips jobs whose laxity is hopelessly low.

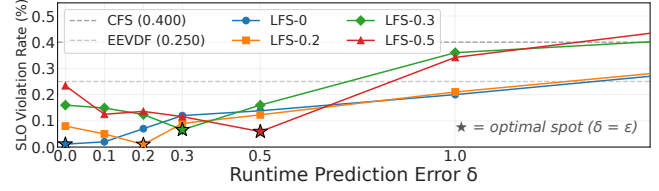


Figure 11: SLO violation rate of policies synthesized under different error bounds ϵ (LFS- ϵ) as the runtime prediction error δ varies. $\star$ marks $\epsilon = \delta$.

for triage, and EDF not at all.

Second, the **objective structure** determines the policies' scheduling priorities, while prediction uncertainty determines their reliance on estimates of future work. Objectives that minimize SLO violations consistently yield urgency-driven policies, with the penalty shape determining which urgency signal dominates. A binary penalty [1, 59, 65] charges a flat cost per miss, letting the scheduler abandon infeasible jobs to greedily save the most deadlines (Triage-SRTF); a stepwise penalty [2, 22] introduces discrete tiers beyond the deadline, making tier containment the goal, and LLF emerges as optimal; and a continuous penalty [39] eliminates tiers, making total tardiness the objective and producing an EDF/SRTF hybrid that orders by deadline while using remaining-time predictions to clear short jobs first. For performance and fairness objectives, policies shift from urgency toward work efficiency and equity: they become SRTF- or SRIF-like variants that exploit compute-I/O overlap, or rely on progress- and service-based signals to equalize allocation across jobs.

7.3 Robustness to Prediction Error

We examine whether LFS synthesizes worst-case optimal policies under the specified prediction-error bounds. Figure 11 evaluates policies synthesized with explicit error bounds (§5) against the actual runtime prediction error δ . We sweep δ from 0 to 1.0 and evaluate four policies synthesized under different assumed bounds ($\epsilon \in \{0, 0.2, 0.3, 0.5\}$). To control prediction error precisely, we run this experiment in a simulator, using synthetic traffic with asymmetric arrival rates and heterogeneous SLOs.¹¹ At each error level, the best policy is the one whose synthesis bound matches the true error ($\epsilon = \delta$, marked by $\star$). Naively trusting predictions (LFS-0) yields near-zero violations under perfect accuracy but degrades sharply as error grows. Conversely, overly conservative policies (e.g., LFS-0.5) avoid catastrophic failure at high error but ignore useful predictive signal, leaving performance on the table when error is low. By calibrating the policy to the actual error, LFS safely exploits imperfect models and consistently outperforms prediction-agnostic CFS and EEVDF.

¹¹Representative workloads include RocksDB and Memcached [17, 45].

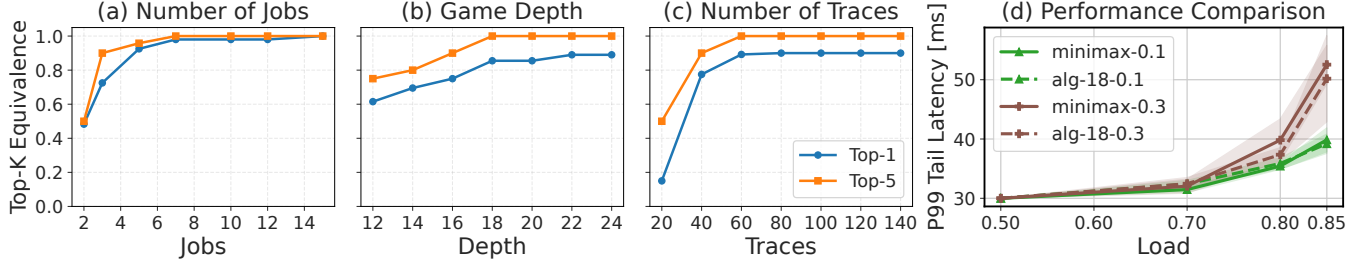


Figure 12: (a)–(c) Sensitivity of the learned policy to simulation configuration. (d) Performance comparison of minimax- ϵ and alg-d- ϵ . Solid and dashed lines denote minimax- ϵ and alg-d- ϵ respectively; shaded bands show $\pm\sigma$.

7.4 Sensitivity to Synthesis Parameters

Because policies are synthesized in simulation, their quality depends on the simulation configuration. We study three parameters: (1) the number of jobs, (2) the planning depth, and (3) the number of simulation traces used to build the decision tree. For each, we measure rank agreement (Top-1 and Top-5) with a policy synthesized under a high-cost reference setting. Figure 12(a)–(c) shows the results.

Num. jobs. Increasing the number of jobs exposes richer global features (e.g., runqueue statistics), but also increases the branching factor and simulation cost. Since the policy is a ranking function, its structure should not depend on the absolute number of jobs once sufficient context is present. We vary the number of jobs from 2 to 15, using 15 as the reference. Agreement improves quickly and saturates at 7 jobs, which we use.

Planning depth. Depth determines the lookahead horizon of the minimax search. Larger depths capture longer-term interactions between jobs, while shallow depths emphasize immediate payoff and lead to myopic decisions. We compare against a deep reference (depth 26). Agreement increases with depth and saturates at 18 (Top-5: 100%, Top-1: 85.5%), with little further gain beyond depth 20. We use depth 18.

Number of simulation traces. More traces improve data coverage but add cost. Using 150 traces as the reference, agreement plateaus beyond 60 (Top-1 $>90\%$), so we use 60. All three parameters show clear saturation; we select the smallest saturating value of each to balance quality and cost.

7.5 Bounded-Depth Optimality Gap

We examine whether the optimality gap introduced by bounded-depth search is small in practice. With the synthetic traffic of §7.3, we measure P99 latency in a discrete-event simulator. Figure 12(d) shows that alg-18- ϵ closely tracks minimax- ϵ across the full load range, with no statistically significant gap at any measured load point. This confirms that depth-18 search incurs minimal optimality loss in practice, consistent with the agreement saturation in §7.4.

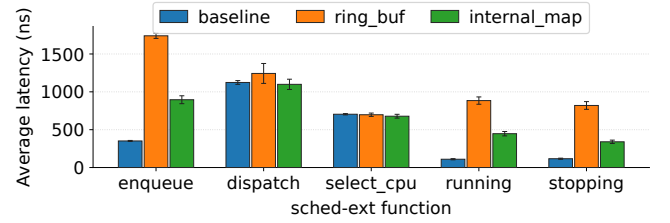


Figure 13: Per-callback latency of the uninstrumented sched_ext scheduler (baseline) and with feature collection via a ring buffer or an eBPF map.

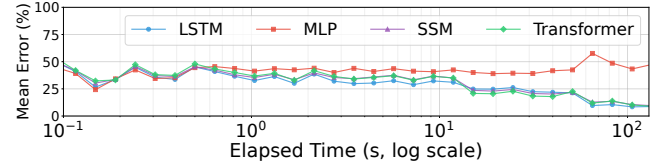


Figure 14: Model prediction error (%) over time.

7.6 Data-Plane Overhead

LFS maintains per-task state for prediction and updates it at each sched_ext callback, adding work to the scheduling path. We examine which data structure minimizes data collection overhead on the scheduling path. We measure callback latency with feature collection via a ring buffer and via an internal eBPF map, and compare both against an uninstrumented sched_ext baseline. As shown in Figure 13, the ring-buffer design incurs substantial overhead, increasing latency by 5–8 $\times$ for enqueue, running, and stopping (up to $\sim 1.7\mu\text{s}$ per callback). The map-based design consistently reduces this overhead, cutting latency by up to 2.4 $\times$ relative to the ring buffer. These results show that map-based collection supports per-task feature updates with lower callback latency than the ring-buffer design, keeping per-callback costs below $1\mu\text{s}$.

7.7 Prediction Accuracy and Inference Cost

The prediction model supplies the latent state on which scheduling decisions depend, and its accuracy determines the error bound under which policies can be synthesized. We examine how model choice affects prediction quality and inference cost. We evaluate four architectures: MLP, LSTM, Transformer, and SSM. Figure 14 shows that the sequence

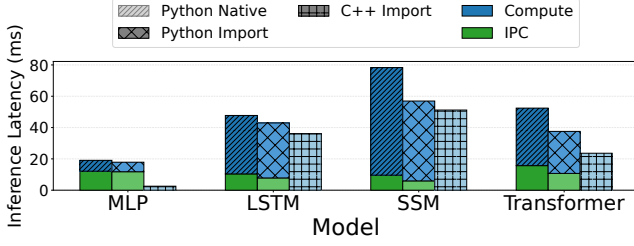


Figure 15: Model inference latency.

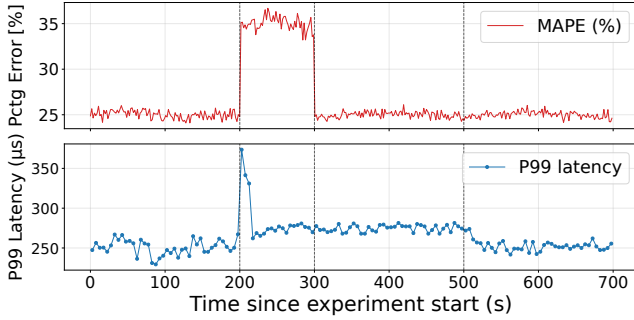


Figure 16: Effect of guardrail on LFS under prediction error: MAE-triggered policy switching and P99 latency over time.

models improve as execution progresses, benefiting from accumulated history, while the MLP, which relies solely on instantaneous features, remains largely unchanged. Models with historical context thus achieve consistently lower error, while differences among the sequence architectures are marginal.

Figure 15 shows inference latency for three deployment options: native Python, PyTorch’s AOTInductor loaded from Python, and LibTorch’s AOTInductor loaded from C++. With batch size 1024, the C++ AOTInductor deployment reduces average inference latency by up to $3.1\times$ relative to native Python, confirming that ahead-of-time compilation and C++ integration are critical for embedding ML models in the scheduler. Given its stable accuracy over time and low inference cost, we adopt the Transformer in our final system.

7.8 Guardrail Under Error Drift

We examine whether the guardrail can recover tail-latency performance when prediction error exceeds the current policy’s bound (Figure 16). We run Nginx (8 workers) under a policy synthesized for tail latency with a 30% error bound (alg-18-0.3). At $T=200$ s, we perturb the predictions to raise MAPE to $\sim 35\%$, which triggers a switch to a more conservative policy (alg-18-0.5). P99 latency briefly spikes but stabilizes quickly after the switch. Once the error stays within the original bound for five consecutive windows (~ 200 s), the system reverts to the original policy at $T=500$ s to recover performance. These results show that the guardrail adapts policy selection to prediction accuracy, restoring stable tail latency after a brief transition.

8 Related Work

General-purpose OS schedulers encode fixed tradeoffs among responsiveness, fairness, and throughput through hand-designed heuristics rather than explicit objectives [55]. Linux CFS approximates fairness via virtual runtime [49], while newer designs such as EEVDF revisit deadline-based scheduling for latency [40, 54]. However, these interfaces expose coarse knobs, forcing operators to indirectly realize objectives through tuning, which is brittle under contention and heterogeneous workloads [7, 37, 38].

Systems for microsecond-scale tail latency use fine-grained preemption and core specialization [18, 29, 33, 48], or enforce isolation under heavy-tailed arrivals [13, 36]. These systems rely on hand-designed policies tuned for specific regimes, and often require dedicated runtimes, kernel bypass, or specialized hardware. Frameworks such as ghOst and sched_ext improve programmability but leave policy design to the developer [15, 27, 51]; LFS builds on the latter.

Learning-based approaches attempt to automate scheduling via trained policies or knob tuning [10, 14, 32, 58, 64], but primarily optimize empirical performance under a fixed workload distribution and provide no guarantees under distribution shift or unmodeled interference. Objective-driven resource allocation has been explored in cluster scheduling [20, 23, 41, 57], highlighting the value of explicit objectives; these systems operate at coarser timescales and do not address fine-grained CPU scheduling or the challenges of integrating learned state into it.

Syntra [50] synthesizes video-streaming controllers from a symbolic environment model, deriving feasible environment actions from observed history. LFS currently uses per-step bounds on prediction errors and contention delays. A Syntra-style extension could model latent CPU state and constraints across jobs and time, using offline quantifier elimination or solver-based projection to restrict adversarial actions to those consistent with the model and observed history. We leave this extension to future work.

9 Conclusion

LFS casts CPU scheduling as an objective-driven synthesis problem, replacing fixed heuristics with policies derived automatically from high-level goals. By combining learned latent state with explicit uncertainty bounds, it produces policies that are robust to prediction error and contention, and its game-theoretic formulation ensures that each synthesized policy is optimal within the induced model. Asynchronous inference over a shared-memory data plane keeps ML off the scheduler’s critical path. Across five objectives and six workloads, synthesized policies match or exceed the best heuristic baselines, and their structure shifts predictably with the objective and with prediction accuracy.

Acknowledgement

This work is supported by the National Science Foundation under Grant No. 2326576 and No. 2422130 and the InfraAI Center at UT Austin.

References

- [1] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I Davis. A Comprehensive Survey of Industry Practice in Real-Time Systems. *Real-Time Systems*, 58(3):358–398, 2022.
- [2] Amazon Web Services. Amazon Compute Service Level Agreement. <https://aws.amazon.com/compute/sla/>. [Accessed 30-01-2026].
- [3] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 929–947, 2024.
- [4] Apache Hadoop. YARN Capacity Scheduler. <https://apache.github.io/hadoop/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html>. [Accessed 30-01-2026].
- [5] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward Formally Verifying Congestion Control Behavior. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 1–16, 2021.
- [6] Romil Bhardwaj, Kirthevasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Cilantro: Performance-Aware Resource Allocation for General Objectives via Online Feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 623–643, 2023.
- [7] Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 85–96, 2018.
- [8] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, 2014.
- [9] Peter Brucker. Scheduling Algorithms. *Journal of the Operational Research Society*, 50:774–774, 1999.
- [10] Jingde Chen, Subho S Banerjee, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. Machine Learning for Load Balancing in the Linux Kernel. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 67–74, 2020.
- [11] Shuang Chen, Christina Delimitrou, and José F Martínez. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.
- [12] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.
- [13] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 621–637, 2021.
- [14] Rohit Dwivedula, Divyanshu Saxena, Aditya Akella, Swarat Chaudhuri, and Daehyeok Kim. Man-Made Heuristics Are Dead. Long Live Code Generators! In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks*, pages 51–60, 2025.
- [15] eBPF.io. Extended Berkeley Packet Filter (eBPF). <https://ebpf.io/>. [Accessed 30-01-2026].
- [16] Stijn Eyerman and Lieven Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3):42–53, 2008.
- [17] Facebook. RocksDB. <https://rocksdb.org/>. [Accessed 30-01-2026].
- [18] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.
- [19] Ron Gabor, Shlomo Weiss, and Avi Mendelson. Fairness Enforcement in Switch On Event Multithreading. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(3):15–es, 2007.

- [20] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.
- [21] Saksham Goel, Benjamin Mikek, Jehad Aly, Venkat Arun, Ahmed Saeed, and Aditya Akella. A Performance Verification Methodology for Resource Allocation Heuristics. *arXiv preprint arXiv:2301.04205*, 2023.
- [22] Google Cloud. Compute Engine Service Level Agreement (SLA). <https://cloud.google.com/compute/sla>. [Accessed 30-01-2026].
- [23] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic Scheduling in Multi-Resource Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 65–80, 2016.
- [24] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. Protean: VM Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861, 2020.
- [25] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S Gunawi. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 173–190, 2020.
- [26] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.
- [27] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghOST: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 588–604, 2021.
- [28] Congfeng Jiang, Yitao Qiu, Weisong Shi, Zhefeng Ge, Jiwei Wang, Shenglei Chen, Christophe Cérin, Zujie Ren, Guoyao Xu, and Jiangbin Lin. Characterizing Co-Located Workloads in Alibaba Cloud Datacenters. *IEEE Transactions on Cloud Computing*, 10(4):2381–2397, 2022.
- [29] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for μ second-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.
- [30] Frank P Kelly, Aman K Maulloo, and David Kim Hong Tan. Rate Control for Communication Networks: Shadow Prices, Proportional Fairness and Stability. *Journal of the Operational Research Society*, 49(3):237–252, 1998.
- [31] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.
- [32] Georgios Liargkovas, Prabhpreet Singh Sodhi, and Kostis Kaffes. Set It and Forget It: Zero-Mod ML Magic for Linux Tuning. In *Proceedings of the 4th Workshop on Practical Adoption Challenges of ML for Systems*, pages 37–43, 2025.
- [33] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E Stephens, Hassan Wassel, and Aditya Akella. RingLeader: Efficiently Offloading Intra-Server Orchestration to NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1293–1308, 2023.
- [34] Linux Foundation. Scheduler Tuning. <https://kernel-internals.org/sched/sched-tuning/>. [Accessed 30-01-2026].
- [35] Jane WS Liu et al. *Real-Time Systems*. Pearson Education India, 2006.
- [36] David Lo, Liquan Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.
- [37] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, A Fedorova, and V Quéma. Your Cores Are Slacking Off—Or Why OS Scheduling Is a Hard Problem. *login: The USENIX Magazine*, 41(4):6–13, 2016.
- [38] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

- [39] Yujia Luo. Time Constraints and Fault Tolerance in Autonomous Driving Systems. *Tech. rep, Tech. Rep.*, 2019.
- [40] LWN.net. An EEVDF CPU Scheduler for Linux. <https://lwn.net/Articles/925371/>. [Accessed 09-03-2025].
- [41] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and Efficient GPU Cluster Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.
- [42] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. Contention-Aware Performance Prediction for Virtualized Network Functions. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 270–282, 2020.
- [43] Huizi Mao, Xiaodong Yang, and William J Dally. A Delay Metric for Video Object Detection: What Average Precision Fails to Tell. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 573–582, 2019.
- [44] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44*, page 248–259, New York, NY, USA, 2011. Association for Computing Machinery.
- [45] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.
- [46] Orange 5G Lab. Smart Stadium – The Stadium’s 5G Revolution. <https://5glab.orange.com/en/realisations/smart-stadium-the-stadiums-5g-revolution/>. [Accessed 30-01-2026].
- [47] Martin J Osborne and Ariel Rubinstein. *A Course in Game Theory*. MIT press, 1994.
- [48] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.
- [49] Chandandeep Singh Pabla. Completely Fair Scheduler. *Linux Journal*, 2009(184):4, 2009.
- [50] Jia Pan, Anup Agarwal, Işıl Dillig, and Venkat Arun. Syntra: Synthesizing Cross-Layer Controllers for Low-Latency Video Streaming. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, 2026.
- [51] sched_ext. sched-ext/scx: sched_ext Schedulers and Tools. <https://github.com/sched-ext/scx>. [Accessed 30-01-2026].
- [52] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E Gonzalez, and Ion Stoica. Fairness in Serving Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 965–988, 2024.
- [53] SPEC. SPEC CPU 2017 Benchmarks. <https://www.spec.org/cpu2017/>. [Accessed 30-01-2026].
- [54] Ion Stoica and Hussein Abdel-Wahab. Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation. *Old Dominion Univ., Norfolk, VA, Tech. Rep. TR-95-22*, 1995.
- [55] Andrew S Tanenbaum and Albert S Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, 1997.
- [56] George Tychogiorgos, Athanasios Gkelias, and Kin K Leung. Utility-Proportional Fairness in Wireless Networks. In *2012 IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, pages 839–844. IEEE, 2012.
- [57] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.
- [58] Xinbo Wang, Shian Jia, Ziyang Huang, Jing Cao, and Mingli Song. Mixture-of-Schedulers: An Adaptive Scheduling Agent as a Learned Router for Expert Policies. *arXiv preprint arXiv:2511.11628*, 2025.
- [59] WebRTC. Real-Time Networking. <https://webrtcforthe curious.com/docs/05-real-time-networking/>. [Accessed 30-01-2026].

- [60] Dingyu Yang, Ziyang Xiao, Dongxiang Zhang, Shuhao Zhang, Jian Cao, and Gang Chen. PREACT: Predictive Resource Allocation for Bursty Workloads in a Co-located Data Center. In *Proceedings of the 53rd International Conference on Parallel Processing*, pages 722–731, 2024.
- [61] Mingyan Yang, Guanjie Wang, Manqi Luo, Yifei Liu, Chen Chen, Han Zhao, Yu Feng, Quan Chen, and Minyi Guo. Justitia: Fair and Efficient Scheduling for LLM Applications. *arXiv preprint arXiv:2510.17015*, 2025.
- [62] Xiao Zhang and Daehyeok Kim. Enabling SLO-Aware 5G Multi-Access Edge Computing with SMEC. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pages 1759–1776, 2026.
- [63] Yongkang Zhang, Yinghao Yu, Wei Wang, Qiukai Chen, Jie Wu, Zuowei Zhang, Jiang Zhong, Tianchen Ding, Qizhen Weng, Lingyun Yang, et al. Workload Consolidation in Alibaba Clusters: The Good, the Bad, and the Ugly. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 210–225, 2022.
- [64] Yusheng Zheng, Yanpeng Hu, Wei Zhang, and Andi Quinn. Towards Agentic OS: An LLM Agent Framework for Linux Schedulers. *arXiv preprint arXiv:2509.01245*, 2025.
- [65] Zoom. Accessing Meeting and Phone Statistics. https://support.zoom.com/hc/en/article?id=zm_kb&sysparm_article=KB0070504. [Accessed 30-01-2026].

A Example Objective Implementation

This section presents example objectives expressed through LFS’s interface.

A.1 Composite Objective

Figure 17 introduces an unfairness ratio (max slowdown over min slowdown) and applies a penalty only when it exceeds a threshold κ , effectively constraining the system to operate within a bounded fairness regime. The penalty weight λ controls the strength of this constraint relative to the primary performance objective.

A.2 Minimizing Makespan

In Figure 18, we compute makespan as the maximum finish_time across jobs (L3–L5) and return it for the optimizer to minimize (L6).

A.3 Max-min Fairness

Figure 19 implements max-min fairness by lexicographically prioritizing the least-served jobs. We first collect each job’s attained service `svc_time` (L3–L7) and sort the values in ascending order (L9–L10), so that the smallest values correspond to the most under-served jobs. The objective is then constructed lexicographically by maximizing the minimum service, followed by the second minimum as a tie-breaker (L12–L18). Since the synthesis engine minimizes, we negate these values.

A.4 Proportional Fairness

Figure 20 implements proportional fairness (PF) in the CPU scheduling context. We model PF for CPU scheduling by treating each job’s effective CPU share as its allocation variable. For a job i , we define $x_i = s_i / (q_i + s_i)$, where s_i denotes the cumulative CPU service received and $q_i + s_i$ is the job’s observed residency in the system. Intuitively, x_i captures the fraction of time the job has spent executing rather than waiting, and thus serves as a normalized long-term CPU share under time multiplexing. We then maximize $\sum_i \log x_i$, the standard proportional-fair objective [30, 56].

A.5 Deviation-based Fairness

Figure 21 implements deviation-based fairness by penalizing dispersion in attained service across jobs. We first collect the cumulative service time `svc_time` of active or completed jobs (L4–L7) and compute the mean service level (L8). The objective then accumulates the squared deviation of each job’s service from this mean (L9–L14) and returns it (L15), driving the optimizer toward policies that equalize CPU service across jobs.

```
1 fn calculate_metrics(&self) -> SchedulingMetrics {
2   let (lambda, kappa) = (100.0, 3.0); // fairness penalty,
   unfairness bound
3
4   // --- Performance term (SLO penalty) ---
5   let mut sum_penalty = 0.0;
6   for job in &self.jobs {
7     sum_penalty += if job.finish_time <= job.deadline {
8       0.0 } else { 1.0 };
9
10  // --- Fairness constraint (slowdown imbalance) ---
11  let max_slowdown = self.jobs.iter()
12    .map(|j| j slowdown).fold(0.0, f32::max);
13  let min_slowdown = self.jobs.iter()
14    .map(|j| j slowdown).fold(f32::MAX, f32::min);
15
16  let unfairness = max_slowdown / (min_slowdown + 1e-5);
17  let fairness_penalty = lambda * (unfairness - kappa).
18    max(0.0);
19
20  let total = sum_penalty + fairness_penalty;
21  SchedulingMetrics::new([total])
22 }
```

Figure 17: Composite objective with a soft fairness constraint.

```
1 fn calculate_metrics(&self) -> SchedulingMetrics {
2   // Makespan = completion time of the last finished job.
3   let makespan = self.jobs.iter()
4     .map(|j| j.finish_time)
5     .fold(0.0, f32::max);
6   SchedulingMetrics::new([makespan])
7 }
```

Figure 18: Objective that minimizes makespan.

A.6 Interface Limitations

Not all scheduling objectives are expressible under the current feature set and payoff interface. We discuss a few classes of objectives that fall outside our scope.

Missing features. While the current feature set captures signals for many common objectives, it does not support those requiring additional abstractions or finer-grained measurements. For example, tenant-level fairness requires grouping primitives (e.g., `group_id`) to aggregate across related jobs, and power-aware objectives require energy or power telemetry, which is not currently exposed. Supporting such objectives requires extending the feature vocabulary and building the needed accounting and instrumentation.

Per-step constraints. Because our framework optimizes a global payoff over a horizon, it cannot enforce strict per-step constraints. A representative example is instantaneous fairness, which requires a constraint to hold at every scheduling step [4, 8, 26]. While proxy objectives—such as minimizing the number of fairness violations over a horizon—can be introduced, they only approximate the instantaneous constraint.

```

1 fn calculate_metrics(&self) -> SchedulingMetrics {
2   // Max-min fairness over attained service.
3   // 1. collect attained service of active/completed jobs.
4
5   let mut services: Vec<f32> = self.jobs.iter()
6     .filter(|j| j.runnable || j.in_io || j.finish_time >
7       0.0)
8     .map(|j| j.svc_time)
9     .collect();
10
11   // 2. sort in ascending order (smallest first).
12   services.sort_by(|a, b| a.partial_cmp(b).unwrap());
13
14   // 3. lexicographic objective:
15   // maximize minimum service, then second minimum, etc.
16   // Since the engine minimizes, negate values.
17   SchedulingMetrics::new([
18     -services[0], // maximize min service
19     -services.get(1).copied().unwrap_or(0.0), // tie-
20       breaker
21   ])
22 }

```

Figure 19: Max-min fairness objective written lexicographically over attained service.

```

1 fn calculate_metrics(&self) -> SchedulingMetrics {
2   // Proportional-fair utility over each job's CPU share.
3   let eps = 1e-9_f32;
4   let mut utility = 0.0_f32;
5   for job in &self.jobs {
6     let svc = job.svc_time.max(0.0);
7     let residency = (job.qdelay + job.svc_time).max(eps);
8     ;
9     let cpu_share = (svc / residency).max(eps);
10    utility += cpu_share.ln();
11  }
12  // Negate because the synthesis engine minimizes.
13  SchedulingMetrics::new([-utility])
14 }

```

Figure 20: Proportional fairness objective example for CPU allocation.

B System Data Characteristics

Input features from the OS span very different ranges: event counters are typically 10^3 – 10^4 , whereas cumulative execution-time fields are in nanoseconds (10^9 – 10^{12}). Fields also differ in their dynamics: most counters increase monotonically, some are roughly stationary (e.g., working set size), and others are event timestamps (e.g., sleep_start, io_start). The prediction targets (remaining burst time) also vary in scale across applications: Nginx bursts are always below 1 ms, stress-ng bursts exceed 5 ms, and DBBench and Redis fall in between.

C Prediction Models

To ensure a fair comparison, we constrain all models to have comparable parameter counts. All models are capped at 150K

```

1 fn calculate_metrics(&self) -> SchedulingMetrics {
2   // minimize dispersion in attained service.
3   // Here we use squared deviation from the mean.
4   let services: Vec<f32> = self.jobs.iter()
5     .filter(|j| j.runnable || j.in_io || j.finish_time >
6       0.0)
7     .map(|j| j.svc_time)
8     .collect();
9   let mean = services.iter().sum::<f32>() / services.len()
10     as f32;
11   let deviation = services.iter()
12     .map(|x| {
13       let d = x - mean;
14       d * d
15     })
16     .sum::<f32>();
17   SchedulingMetrics::new([deviation])
18 }

```

Figure 21: Deviation-based fairness objective that minimizes variance in attained service.

parameters. Each model is trained for 50 epochs on data collected from 142 runs spanning diverse application mixes (dbbench, nginx, redis). We minimize mean absolute error (MAE) loss using a cosine-annealing learning-rate schedule (initial learning rate 10^{-3}) and batch size 16384 on 4 GPUs. To stabilize training across workloads with differing burst-length distributions, we apply $\log(1 + x)$ normalization followed by min–max scaling to map the targets to $[0, 1]$.

D Broader Use Cases

LFS’s approach of combining user-defined objectives, learned state, and policy synthesis under bounded uncertainty could extend to other scheduling settings. For edge scheduling [62], the payoff could measure end-to-end SLO violations, with the model accounting for uplink and downlink delays alongside computation. For LLM serving [52], the payoff could capture fairness in service measured by input and output token costs, while the model accounts for unknown output lengths, batching, and GPU memory limits. Cloud resource management [6, 24] could use placement and allocation actions to optimize SLO satisfaction, utilization, or cost. Each extension requires observable signals for estimating the relevant state, valid uncertainty bounds, and an execution model and action space that permit tractable synthesis; the predictors, scheduling model, and runtime integration would need to be adapted to each setting. We leave these extensions to future work.