

OS-Evolve

An Evolutionary Framework for Automating CPU Scheduling

Yoga Sri Varshan Varadharajan
UT Austin
yogasrivarshan@utexas.edu

Krishanu Saini
UT Austin
krishanu.saini@utexas.edu

Abstract

Kernel policies in mature operating systems like Linux have been battle-tested over decades to provide the best average-case performance. However, modern datacenter environments often require specialized policies—optimizing for microsecond-scale tail latency or high-throughput batch processing—that generic schedulers like CFS fail to provide efficiently. We propose OS-Evolve, an automated framework that leverages Large Language Models (LLMs) and evolutionary algorithms to generate, mutate, and optimize C++ scheduling policies. By utilizing the Google ghOSt userspace scheduling framework, we bypass the operational hazards of kernel development. This proposal outlines our results in evolving policies that outperform CFS across varying thread and core configurations and details our implementation of a Global Knowledge Base (GKB). Our proposed extension focuses on bridging the gap between offline evolution and online execution, enabling a self-adapting OS that monotonically improves by hot-swapping ‘veteran’ policies matched to real-time workload signatures.

1. Motivation and Problem Statement

1.1. The Limits of ‘One Size Fits All’

The Linux Completely Fair Scheduler (CFS), introduced in 2007 [1], is designed as a ‘best average case’ algorithm. It excels at fairness across generic tasks. However, one can argue that statistically best algorithms are not the best performing ones on specific data loads. For instance, a First-Come-First-Serve (FCFS) approach performs poorly if the CPU is clogged with high-compute tasks at the front of the queue, creating Head-of-Line (HoL) blocking.

Modern datacenters often feature repeating, predictable workloads—either short interactive bursts or long computation cycles. Specialized handcrafted schedulers like Shinjuku [4] and Concord [3] have demonstrated that tailoring logic to these specific patterns can yield massive gains in tail

latency and throughput. However, these systems are manually designed, requiring deep domain expertise.

1.2. Operational Challenges

Defining heuristics for new workloads is an intricate, labor-intensive job involving trace crunching, simulation, and manual signal tracking. Furthermore, implementing these systems in the kernel is operationally expensive; one cannot reboot a production server every time a policy requires an update.

Goal: We propose to solve these challenges by gluing together the Google ghOSt framework [2] (which acts as an actuator for delegating kernel tasks to userspace) with an LLM-driven evolutionary engine.

2. Proposed Architecture: OS-Evolve

Our framework treats the CPU scheduler not as a black-box neural network, but as a ‘White Box’ C++ program that can be iteratively improved.

2.1. Evolutionary Engine

Building on the ShinkaEvolve [5] paradigm, our engine manages a population of scheduler implementations.

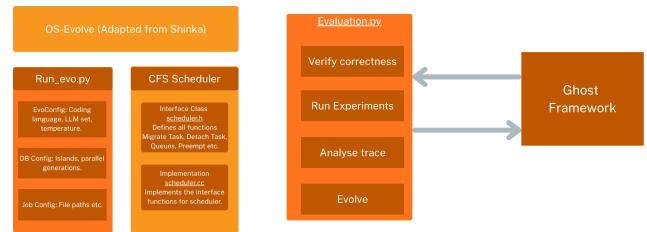


Figure 1. OS-Evolve Architecture. The evolutionary loop drives the LLM to mutate C++ code defined in `scheduler.cc`. These are compiled against ghOSt, evaluated on benchmarks, and scored to drive the next generation.

- **Input:** A baseline C++ scheduler (simple CFS implementation).

- **Mutation:** The LLM is prompted to modify blocks of code using boundary marker comments (e.g., 'Evolve-Block-Start', 'Evolve-Block-End').
- **Evaluation:** Candidates are compiled into a ghOSt agent and subjected to various CPU/IO/Memory bound workloads.

3. Results

We have conducted experiments to validate the feasibility of evolving C++ scheduler code. We segregated our testing into two categories: Head-of-Line (HoL) blocking tests and standard SysBench CPU workloads.

3.1. Pareto Analysis (HoL Testing)

To test the scheduler's ability to handle queue blocking, we designed a custom synthetic workload with fast and slow threads. Figure 2 illustrates the Pareto Frontier of Latency vs. Throughput.

- **Observation:** The evolutionary process successfully pushes the boundary of performance on the 4-thread HoL workload.
- **Performance:** Specific generations (Gen 5 and 8) significantly improved throughput by prioritizing fast threads, effectively mitigating the HoL blocking phenomenon.

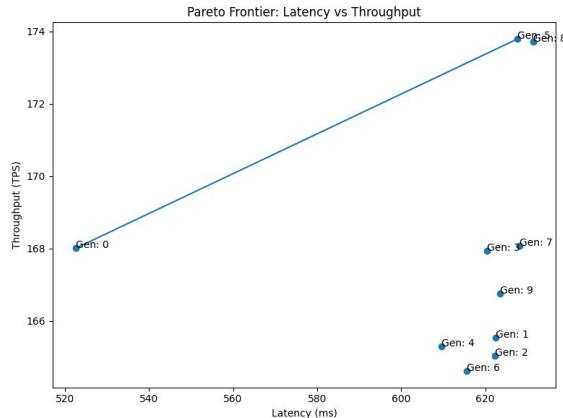


Figure 2. Pareto Frontier of Latency vs. Throughput for HoL tests. Higher generations move towards the top-right, indicating improved throughput over the baseline.

3.2. SysBench Benchmarking Across Workloads

We extended our evaluation to standard 'SysBench' CPU workloads, testing across varying thread counts and CPU allocations. This tests the scheduler's ability to adapt to different levels of concurrency and resource availability. Table 1 summarizes the improvements.

The results indicate that as the complexity of the scheduling decision increases (more threads or more cores),

Num Threads	Num CPUs	Initial Score (TPS)	Best Evolved Score
4	5	1234.56	1240.98
32	5	1504.80	1517.00
32	10	4621.63	4737.81
64	5	1502.34	1541.11

Table 1. Comparison of Throughput (Events/Sec) across different SysBench workloads. OS-Evolve consistently discovers policies that outperform the initial baseline, with significant gains observed in high-concurrency scenarios (32 threads/10 CPUs and 64 threads/5 CPUs).

the evolutionary framework finds greater optimization opportunities. For instance, in the 32 Threads / 10 CPUs scenario, the evolved policy achieved a substantial jump from 4621.63 to 4737.81 events/sec.

3.3. Analysis of Evolved Heuristics

Our experiments revealed a hierarchical evolution of logic:

1. **Lower Order (Hyperparameters):** Early generations focused on tuning constants, such as changing load imbalance thresholds to favor stickiness.
2. **Higher Order (Algorithmic):** Advanced generations (e.g., Gen 23) implemented complex logic.
 - **Lockless Logic:** Implementing lockless runqueue load checks (`rq size + oncpu`) in '`SelectTaskRq`' to avoid cross-CPU lock contention.
 - **Tiered Placement:** Correctly treating CPUs with running tasks as non-idle to prevent cache thrashing.

4. Proposal Extension: Offline-Online Integration

We have already implemented the Global Knowledge Base (GKB) as a mechanism to store and retrieve 'veteran' policies. The core focus of this proposal extension is to operationalize the GKB into a robust Offline-Online System for real-world deployment.

4.1. The Global Knowledge Base (Implemented)

A limitation of evolutionary approaches is that it does not remember past evolutions efficiently. Complex reusable logic is often rediscovered from scratch. To solve this, we implemented the GKB (Fig. 3) using a SQLite backend.

- **Function:** It acts as a permanent archive for 'veteran' policies—schedulers that have statistically proven themselves on specific workloads.
- **Monotonic Improvement:** Before a new evolution run, OS-Evolve queries the GKB for the best existing policy to use as a 'parent,' ensuring the system improves monotonically over time.

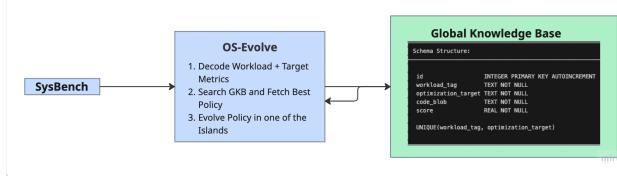


Figure 3. The Global Knowledge Base workflow. The GKB (center) archives the best policies.

4.2. Proposed: Decoupled Execution Loop

Our extension aims to separate the heavy lifting of the evolutionary algorithm from fast CPU scheduling to create a self-adapting OS:

- 1. Online (Hot-Swap):** When a new workload arrives (e.g., identified via trace signatures), the userspace agent queries the GKB. It retrieves the optimal policy (e.g., ‘Throughput-Veteran-Gen23’) and hot-swaps the logic without kernel modification.
- 2. Offline (Feedback):** Metrics from the live run (SLO violations, latency spikes) trigger a corresponding offline evolution session.
- 3. Closure:** The result of this evolution is inserted back into the GKB, creating a continuous feedback loop as shown in Figure 4

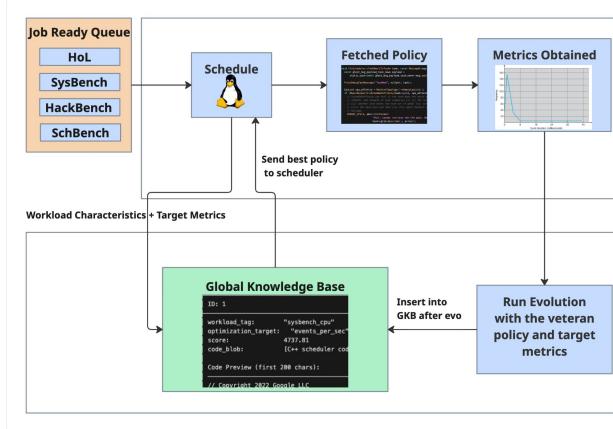


Figure 4. The proposed extension involves closing the loop between the Offline Evolution (bottom) and the Online Scheduler (top).

5. Project Milestones

- Phase 1 (Completed):** Validated OS-Evolve framework and demonstrated throughput gains on SysBench/HoL benchmarks.
- Phase 2 (Completed):** Implemented the Global Knowledge Base (GKB) and the logic to ‘seed’ evolution with veteran policies.

- Phase 3 (Proposed Extension):** Develop the ‘Workload categorization’ module and the Online-Offline feedback loop to automatically classify jobs and trigger hot-swaps.
- Phase 4 (Future):** Focus on Explainability—analyzing evolved code to understand *why* certain heuristics work for specific loads.

6. Conclusion

Handcrafted schedulers like Shinjuku show the power of specialization, but they are static. OS-Evolve proposes a dynamic, self-learning system. By combining the safety of ghOST with the creativity of LLMs and the memory of a Global Knowledge Base, we aim to build an OS that doesn’t just run workloads, but evolves to run them better.

References

- [1] Wei-Cong Fan, Chee-Siang Wong, Wai-Kong Lee, and Seong-Oun Hwang. Comparison of interactivity performance of linux cfs and windows 10 cpu schedulers. In *2020 International Conference on Green and Human Information Technology (ICGHIT)*, pages 31–34. IEEE, 2020. 1
- [2] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombach, 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. 1
- [3] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 466–481, 2023. 1
- [4] Kostis Kaffles, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for {μsecond-scale} tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019. 1
- [5] Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025. 1