SaaKM : Schedulers as Kernel Module*

1st Baptiste Pires
LIP6
Sorbonne Université
Paris France
baptiste.pires@lip6.fr

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

4th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

5th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

***Abstract*—Scheduling plays a critical parts in applications performance and must not be looked upon. However, writing schedulers can be a long and tedious task. There are efforts made over the past years to provide APIs and frameworks to ease the process of writing scheduling policies. In this paper we present SaaKM, a small kernel APIs to write scheduling policies as Linux kernel modules. Our goal is to remove the barriers to write schedulers. We compare our solution to the recently merged ext scheduling class. SaaKM performs simlilarly using a per-cpu FIFO scheduler.**

***Index Terms*—kernel, scheduler, module**

I. Introduction

CPU scheduling is a fundamental aspect of operating systems that directly impacts both system performance and security. Modern computing environments present unprecedented challenges to schedulers: hardware has grown increasingly complex with heterogeneous architectures, NUMA topologies, and hybrid core designs, while applications demonstrate high concurrency patterns with hundreds or thousands of threads [1]. The Linux kernel addresses these challenges through its general-purpose scheduler, the Earliest Eligible Virtual Deadline First (EEVDF) algorithm [2], which replaced the Completely Fair Scheduler (CFS) [3] in version 6.6. While EEVDF represents a significant improvement over CFS, particularly for latency-sensitive workloads, its general-purpose design necessitates trade-offs that prevent optimal performance for specialized workloads [4], [5].

The complexity of implementing custom schedulers in Linux presents substantial barriers to innovation. Developing a scheduler requires deep understanding of kernel internals, complex synchronization mechanisms including RCU and memory barriers, and intricate interactions with multiple kernel subsystems. The multicore nature of modern systems introduces additional challenges around concurrency control, load balancing, and cache coherency that must be carefully managed. Moreover, traditional scheduler development involves a time-consuming cycle of kernel recompilation and system reboot for each iteration, making testing and debugging particularly challenging. Errors in scheduler implementations can lead to system crashes or data corruption, further complicating development efforts.

Recognizing these challenges, the systems community has developed several approaches to simplify scheduler development. These solutions can be broadly categorized into two paradigms: userspace delegation and in-kernel extensibility. Userspace approaches, exemplified by ghOSt [4] and Skyloft [6], delegate scheduling decisions to userspace processes while maintaining a minimal kernel component for enforcement. This approach enables rapid development and testing but introduces communication overhead and potential reliability concerns. In-kernel extensibility approaches maintain scheduling logic within kernel space while providing higher-level programming interfaces. The recently merged ext scheduling class [7] (for the sake of simplicity we will call it sched_extfrom now on) in Linux 6.12 allows developers to implement schedulers as eBPF [8] programs, providing safety guarantees through the eBPF verifier while enabling dynamic loading and unloading. Similarly, Enoki [9] leverages Rust's safety features to implement schedulers as kernel modules with memory safety guarantees.

Despite these advances, existing solutions impose significant constraints on developers. Solutions like sched_ext require mastering eBPF programming and its associated toolchain, while the eBPF verifier's restrictions can limit access to kernel data structures and complex control flow. Rust-based approaches like Enoki require learning a new programming language and its ecosystem. Userspace solutions introduce architectural complexity and potential performance overhead through kernel-userspace communication. These constraints create barriers that may discourage developers from experimenting with scheduling innovations.

Identify applicable funding agency here. If none, delete this.

To address these limitations, we propose SaaKM (Schedulers as a Kernel Module), a framework that prioritizes developer accessibility while maintaining the performance and safety characteristics of in-kernel scheduling. SaaKM allows developers to implement schedulers using familiar C programming and standard Linux kernel module development practices, eliminating the need to learn new languages or specialized subsystems. The framework abstracts complex kernel interactions through a well-defined event-driven API, which simplifies development by clearly distinguishing thread lifecycle events from core management events.

SaaKM introduces a comprehensive thread state machine that provides clear scheduling semantics while enabling runtime validation of state transitions for enhanced debugging. The framework supports multiple concurrent scheduling policies within the same system, allowing applications to select the most appropriate scheduler for their specific requirements. This flexibility enables fine-grained optimization strategies where different workload types can coexist under different scheduling policies simultaneously.

The key contributions of this work are:

- A novel framework for implementing Linux schedulers as kernel modules that maintains compatibility with existing kernel development workflows
- An event-driven API abstraction that simplifies scheduler implementation by clearly delineating scheduling events and hiding synchronization complexity
- A comprehensive thread state machine with runtime validation for enhanced debugging and development experience
- Support for multiple concurrent scheduling policies with per-application policy selection
- A performance evaluation comparing SaaKM with sched_ext across various workloads and use cases

This paper is structured as follows. We first examine current approaches to scheduler development and their limitations. We then present the design and implementation of SaaKM, detailing its architecture and key abstractions. Following this, we evaluate SaaKM's performance and usability compared to existing solutions, particularly sched_ext. Finally, we discuss the implications of our work and future research directions in scheduler development frameworks.

II. Background and Related Work

The Linux scheduler has undergone significant evolution to address the increasing complexity of modern computing environments. Understanding the current landscape of scheduler development approaches is essential to position our contribution effectively. This section examines existing solutions for custom scheduler development, categorizing them based on their execution paradigm and analyzing their respective trade-offs.

A. *Evolution of Linux Schedulers*

The Linux kernel's default process scheduler transitioned from the Completely Fair Scheduler (CFS) to the Earliest Eligible Virtual Deadline First (EEVDF) algorithm in version 6.6, released in 2024. EEVDF was first introduced in a scientific publication in 1995 and aims to distribute CPU time equally among all runnable tasks with the same priority by assigning virtual run times and calculating virtual deadlines for scheduling decisions. While EEVDF represents a significant improvement over CFS, particularly for latency-sensitive workloads, its general-purpose design necessitates trade-offs that prevent optimal performance for specialized applications.

The traditional approach to scheduler modification involves directly implementing a new scheduling class within the kernel. The Linux scheduler architecture utilizes an extensible hierarchy of scheduling classes, where each class encapsulates specific scheduling policies and is handled by the scheduler core. However, this approach requires deep understanding of complex kernel internals, including synchronization mechanisms like RCU and memory barriers, making it accessible only to experienced kernel developers.

B. *In-Kernel Scheduler Frameworks*

Recent years have witnessed significant innovation in frameworks that enable custom scheduler development while maintaining execution within kernel space. These approaches aim to provide higher-level abstractions while preserving the performance characteristics of in-kernel scheduling.

sched_ext: eBPF-Based Schedulers The sched_ext framework, merged into Linux 6.12, allows scheduling policies to be implemented as eBPF programs. sched_ext exports a full scheduling interface through BPF struct_ops, enabling any scheduling algorithm to be implemented while maintaining system integrity through the eBPF verifier. The framework provides safety guarantees by automatically reverting to the default scheduler when errors are detected or runnable tasks stall.

The BPF verifier ensures that custom schedulers have neither memory bugs nor infinite loops, and the framework can update schedulers without kernel reinstallation or system reboot. However, eBPF's restrictions limit access to certain kernel data structures and complex control flow, while requiring developers to master eBPF programming and its associated toolchain.

Enoki: Rust-Based Schedulers Enoki [9] leverages Rust's safety features to implement schedulers as kernel modules. The framework provides memory safety guarantees while offering live update mechanisms that allow upgrading running schedulers without unloading previous versions. Enoki implements a record and replay mechanism for debugging scheduling events, though it requires developers to learn Rust and its ecosystem.

Plugsched: Live Scheduler Updates Plugsched addresses the challenge of scheduler updates by modularizing the Linux scheduler subsystem. The approach uses boundary analysis to extract the scheduler from kernel code into a separate module, enabling dynamic replacement of the scheduler in running systems. Plugsched uses data rebuild techniques to migrate state from old to new schedulers, achieving downtime of less than tens of milliseconds. However, the approach requires careful boundary analysis and imposes constraints on data structure modifications.

C. Userspace Delegation Frameworks

An alternative paradigm delegates scheduling decisions to userspace processes while maintaining a minimal kernel component for enforcement. These approaches enable rapid development and testing but introduce architectural complexity and potential performance overhead.

ghOSt: Userspace Agents Google's ghOSt provides general-purpose delegation of scheduling policies to userspace processes in a Linux environment, offering state encapsulation, communication, and action mechanisms that allow complex expression of scheduling policies within userspace agents. The framework supports policies for various scheduling objectives from microsecond-scale latency to throughput and energy efficiency, with many policies requiring only a few hundred lines of code.

ghOSt's architecture separates scheduling logic from kernel interaction components through agents that run policies in userspace and communicate scheduling decisions to the kernel scheduling class via message queues and transactions. The framework supports rebootless upgrades and recovers from scheduler failures without triggering kernel panics by falling back to CFS when issues occur. Despite these advantages, ghOSt introduces communication overhead between kernel and userspace components and requires managing complex state synchronization.

Skyloft: Userspace Interrupts Skyloft [6] takes a different approach by isolating CPU cores exclusively for userspace schedulers. The framework achieves microsecond-scale scheduling preemption by leveraging userspace interrupts (UINTR), bypassing kernel involvement for interrupt delivery. While this approach minimizes kernel overhead, it requires dedicated hardware support and CPU core isolation.

D. Analysis and Limitations

Each existing approach presents significant trade-offs that may discourage scheduler innovation:

Complexity Barriers: sched_ext requires mastering eBPF programming and its limitations, while Enoki demands learning Rust. Traditional kernel development requires extensive knowledge of complex synchronization mechanisms and kernel internals.

Performance Concerns: Userspace approaches like ghOSt introduce communication overhead between kernel and userspace components. While the overhead is often acceptable, it may be problematic for latency-critical applications.

Flexibility Constraints: eBPF's verifier restrictions can limit scheduler implementations, while approaches like Plugsched impose constraints on data structure modifications. Traditional kernel development requires lengthy recompilation and deployment cycles.

Deployment Complexity: Some frameworks require significant infrastructure changes, custom kernel builds, or specialized build toolchains that may not be readily available in production environments.

E. Our Contribution

The analysis of existing frameworks reveals a gap for approaches that combine the familiarity of traditional kernel module development with the flexibility and safety of modern scheduler frameworks. Our work addresses this gap by providing a framework that:

- Uses familiar C programming and standard Linux kernel module development practices
- Provides clear abstractions that hide complex kernel synchronization mechanisms
- Supports rapid development and testing cycles without requiring specialized toolchains
- Enables multiple concurrent scheduling policies with per-application policy selection
- Maintains compatibility with existing kernel development workflows

This approach aims to lower the barrier to scheduler experimentation while maintaining the performance characteristics and safety guarantees necessary for production deployment.

III. Design

This section presents the design of SaaKM (Schedulers as a Kernel Module), our framework for implementing custom Linux schedulers as kernel modules. We begin by establishing our design goals, then present the overall architecture, and finally detail the key implementation decisions that enable accessible scheduler development while maintaining safety and performance.

A. Design Goals

The complexity of modern scheduler development in Linux presents significant barriers to innovation and experimentation. Our analysis of existing approaches reveals three fundamental challenges that SaaKM aims to address.

Accessibility The primary barrier to scheduler innovation is the steep learning curve required for Linux kernel development. Traditional scheduler implementation requires deep understanding of complex synchronization mechanisms including RCU, memory barriers, and intricate locking protocols. The scheduler has many data structures that are protected by RCU, and incorrect handling of these mechanisms can lead to subtle bugs or system crashes. Existing solutions either require learning new programming paradigms (eBPF for sched_ext, Rust for Enoki) or impose architectural complexity (userspace delegation in ghOSt). SaaKM addresses this by providing a familiar development environment using standard C and Linux kernel module practices, allowing developers to leverage existing knowledge and toolchains.

Performance Scheduling decisions occur on critical system paths where even small overheads can significantly impact overall system responsiveness. Modern Linux kernels rely on highly optimized code paths for networking, security, virtualization, and process management, making it essential that any scheduler framework introduces minimal overhead. SaaKM achieves this by implementing a thin abstraction layer (less than 1500 lines of code) that provides essential services without compromising the performance characteristics of in-kernel scheduling.

Development Velocity Traditional kernel scheduler development involves lengthy cycles of compilation, deployment, and testing. Debugging scheduler implementations is particularly challenging due to the difficulty of reproducing timing-dependent issues and the potential for system crashes during development. SaaKM addresses this through kernel module-based development that enables rapid iteration cycles, comprehensive debugging support through existing kernel debugging infrastructure, and runtime state validation to catch common programming errors.

B. Architecture Overview

SaaKM implements a new scheduling class that integrates seamlessly with Linux's existing scheduler hierarchy. Scheduling classes are implemented through the `sched_class` structure, which contains hooks to functions that must be called whenever an interesting event occurs. Figure 1 illustrates how SaaKM positions itself within this hierarchy.

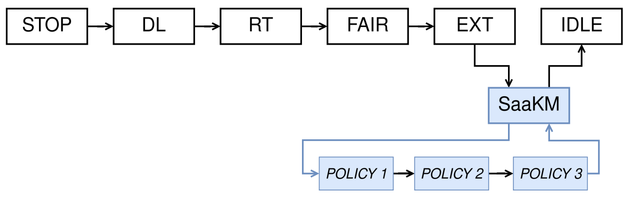

Figure 1. SaaKM scheduling class integration with Linux scheduler hierarchy

Our scheduling class is positioned between the existing ext and idle scheduling classes, ensuring that SaaKM policies are considered only when no higher-priority scheduling classes have runnable tasks.

The framework supports multiple concurrent policies within a single scheduling class, enabling fine-grained specialization where different applications can select optimal schedulers for their specific requirements. This capability addresses the growing recognition that no single scheduling algorithm can optimally serve all workload types in modern computing environments.

C. Event-Driven Abstraction

The core innovation of SaaKM is an event-driven abstraction that simplifies scheduler implementation by clearly delineating scheduling events and hiding the underlying complexity of kernel synchronization mechanisms. Traditional scheduler development requires understanding the multiple code paths that can trigger the same scheduling function. The `enqueue_task` function might lead to process switch, but distinguishing between a newly forked task, a migrating task, and a waking task requires deep knowledge of the calling context.

SaaKM addresses this complexity through a structured event model that categorizes scheduling events into two distinct types:

Thread Events encompass all lifecycle changes for individual tasks, including creation, termination, blocking, and wake-up events. These events provide clear semantics about what is happening to a specific task without requiring the scheduler author to understand the complex code paths that led to the event.

Core Events handle CPU-centric operations such as scheduling decisions, idle transitions, and load balancing activities. These events abstract the complex interactions between the scheduler core and individual CPU runqueues.

Table I provides a comprehensive overview of the event handlers that policies must implement. This event-driven approach ensures that scheduler authors receive clear, actionable information about system state changes without needing to understand the intricate details of kernel synchronization.

Table I
SAAKM EVENT HANDLER FUNCTIONS

Function	Description
Thread Events	
new_prepare(task)	Return CPU ID where newly created task should run
new_place(task, cpu)	Insert new task into specified CPU runqueue
unblock_prepare(task)	Return CPU ID for task transitioning from blocked to ready
unblock_place(task, cpu)	Insert unblocked task into specified CPU runqueue
block(task)	Handle task transitioning to blocked state
yield(task)	Handle voluntary CPU relinquishment
tick(task, cpu)	Process scheduling tick for running task
terminate(task)	Clean up resources for terminating task
Core Events	
schedule(cpu)	Select next task to run on specified CPU
newly_idle(cpu)	Handle CPU transition to idle with no ready tasks
enter_idle(cpu)	Process CPU entering idle state
exit_idle(cpu)	Handle CPU exiting idle state due to runnable task
balancing_select(cpu)	Perform load balancing for specified CPU
core_entry(cpu)	Initialize policy state for CPU coming online
core_exit(cpu)	Clean up policy state for CPU going offline
Policy Management	
init()	Initialize global policy state
free_metadata(task)	Release policy-specific task metadata
checkparam_attr(attr)	Validate scheduler attribute parameters
setparam_attr(attr)	Apply validated scheduler parameters

D. Synchronization Abstraction

One of the most significant challenges in scheduler development is correctly handling the complex synchronization requirements of multicore systems. RCU may need to wake up threads to perform things like completing grace periods and callback execution, creating intricate dependencies between the scheduler and RCU subsystems. Modern Linux kernels use sophisticated synchronization mechanisms including RCU, memory barriers, and hierarchical locking protocols to ensure data consistency across NUMA architectures.

SaaKM abstracts this complexity through several key mechanisms:

Two-Phase Operations For operations that require cross-CPU coordination, SaaKM separates CPU selection from queue manipulation. The `prepare` phase allows policies to make scheduling decisions while holding minimal locks, followed by a `place` phase that performs the actual queue operations with appropriate synchronization. This separation ensures that policies never need to reason about lock ordering or acquisition protocols.

Lock-Free Event Delivery All event handlers are invoked with appropriate locks already held by the SaaKM core. Policies can safely access and modify scheduling state without understanding the underlying locking protocols. This design eliminates common sources of deadlocks and race conditions in scheduler development.

State Transition Validation SaaKM maintains a comprehensive state machine for task scheduling states and validates all transitions at runtime. Figure 2 illustrates the permitted state transitions. This validation catches many common programming errors during development and provides clear diagnostic information when invalid transitions are attempted.

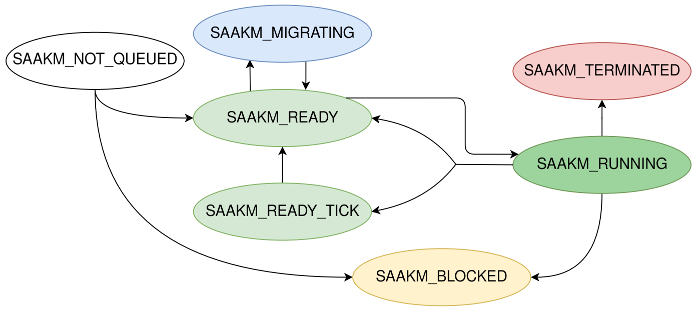

Figure 2. SaaKM thread state machine with permitted transitions

E. Runqueue Management

SaaKM provides three common runqueue implementations to accelerate policy development: FIFO queues for simple round-robin policies, general-purpose linked lists for priority-based scheduling, and red-black trees for algorithms requiring ordered access to tasks. Each implementation handles the necessary metadata management and provides consistent interfaces for task insertion, removal, and traversal operations. Policies must provide ordering functions for list and tree-based runqueues.

F. Load Balancing and Migration

Modern multicore systems require sophisticated load balancing to achieve optimal performance. SaaKM provides infrastructure for both periodic load balancing and idle-initiated balancing through a dedicated software interrupt mechanism.

The framework exports hardware topology information through per-CPU `topology_levels` variables that describe the underlying hierarchy. This information allows policies to make intelligent migration decisions that consider the costs of moving tasks between different levels of the memory hierarchy.

note:*[Task migration is handled transparently through an `OUSTED` flag mechanism, which is fully managed within the scheduling class. Policies do not access this flag directly; instead, the framework ensures correct handling of affinity changes and load balancing operations without requiring policies to track migration state.]*

G. Multi-Policy Support

SaaKM enables multiple scheduling policies to coexist within the same system, allowing applications to select the most appropriate scheduler for their specific requirements. Policies are assigned unique identifiers upon registration and maintained in an ordered list.

When scheduling decisions are required, SaaKM iterates through the registered policies in insertion order, allowing the first policy that has a runnable task to make the scheduling decision. This approach provides flexibility while maintaining low overhead for the common case of a single active policy.

Applications select their preferred scheduling policy using the standard `sched_setscheduler` interface with the `SCHED_SAAKM` policy and a policy-specific identifier in the scheduler attributes. This mechanism integrates seamlessly with existing process management tools and workflows.

H. Debugging and Testing Infrastructure

SaaKM provides comprehensive debugging support through integration with existing kernel debugging infrastructure. Policies can leverage KGDB [10], ftrace, and other standard kernel debugging tools without requiring specialized knowledge of scheduler-specific debugging techniques.

The framework provides configurable runtime validation of state transitions, runqueue consistency checks, and policy-specific invariants. When violations are detected, SaaKM can generate detailed diagnostic information and optionally trigger kernel panics to provide immediate feedback during development.

A procfs interface exposes information about currently loaded policies, their runtime statistics, and configuration parameters. This interface facilitates both development debugging and production monitoring of scheduler behavior. The sysfs interface provides runtime control over debugging features, allowing developers to enable or disable specific validation checks, logging levels, and diagnostic output without recompiling the kernel or reloading policies.

IV. Evaluation

In this section, we evaluate the usability and performance of SaaKM. We present a minimal FIFO scheduler implemented both with SaaKM and the ext scheduler [7], which was recently merged into Linux kernel v6.12. We then compare the performance of both implementations across a set of benchmarks.

We chose to compare against the ext scheduler because it is the most recent kernel-integrated framework, whereas other solutions did not meet our requirements. For example, Enoki [9] targets Linux v5.11, which is 21 releases and 1224 commits behind v6.12 (specifically in the kernel/sched/ directory). Significant changes, such as the EEVDF merge, have affected the scheduler since then. Our focus is on comparing SaaKM to kernel-level scheduler frameworks rather than userspace solutions, which serve different purposes; therefore, we do not include ghOSt, which also targets kernel v5.11, in our evaluation.

A. Experimental Setup

Benchmark platform All experiments were conducted on a dedicated server running Debian 12 (Bookworm) with a custom-patched Linux kernel version 6.12. The system is equipped with an Intel(R) Core(TM) i9-10900K CPU (10 cores, 20 threads via SMT), 20MB L3 cache, and 64GB RAM. This configuration ensures sufficient computational resources and memory bandwidth to reliably evaluate scheduler performance under representative workloads.

Benchmarks We evaluate scheduler performance using a diverse set of workloads from the Phoronix Test Suite [11] and the hackbench [12] microbenchmark. Each benchmark is executed 50 times, with kernel caches cleared between runs to ensure result consistency. We report the mean and 95% confidence interval for all measurements.

Evaluated schedulers To facilitate a controlled comparison, we implement a minimal FIFO scheduler using both SaaKM and sched_ext. Both implementations are designed to make identical scheduling decisions, isolating the impact of the underlying framework. Each scheduler maintains a per-CPU local FIFO runqueue, and threads are statically assigned to CPUs based on their PID. No load balancing is performed, ensuring a deterministic scheduling behavior and minimizing confounding factors. Only benchmark threads are managed by the evaluated policies.

B. Results evaluation

On average, we see 0.33% (+/- 1.04%) of gain with the SaaKM implementation. Table II sums up the results per benchmark. We have a maximum loss with clickhouse for which we avec a decrease of 1.23% of Queries Per Minute (QPM). The application showing the best results is x265 with a gain of 2.48% of Frames Per Second (FPS). Out of the 16 benchmarked applications, 10 have a gain or loss between -0.5% and 0.5%. Only two applictions showed more than 1% loss, the 7zip compression and clickhouse, with respectively -1.2% and -1.23%. **note:***[j'ai des résultats clickhouse avec +0.57% quand le cache est froid et -0.57% au second run, là les résultats c'est le 3ème run, je trouve ça plus pertinent de montrer les résultats avec le cache chaud. Mais on peut peut être se servir des deux autres pour montrer que c'est peut être un problème de latence dans saaKM ?]*

C. Limitations

While our evaluation demonstrates that SaaKM is a practical framework for implementing schedulers, it is limited to a simple per-CPU FIFO policy. Further investigation is needed with more complex schedulers, such as Shinjuku [13] and EEVDF [2], to fully assess SaaKM's capabilities and performance in diverse scenarios. Additionally, our experiments did not utilize SaaKM's multi-policy support. Exploring its impact on workloads involving multiple applications, as well as any potential overhead introduced by policy iteration, remains an open area for future work.

x265, Cassandra and npb shows the best improvements, all three gaining more than 1.99%.

- Definir les métriques
- Definir ce à quoi on se compare
- Présenter l'env et les benchmarks et les motiver
- Présentation et analyse des résultats

note:*[pour le tableau, est-ce que je laisse comme ça ou je remets les pourcentage sans prendre en compte lower/higher is better et je rajoute une petite colonne pour indiquer la lecture de la ligne ? (lower/higher)]*

V. Future Works

Scheduling de policy / ajout de prio entre policy ?

Rajouter une API pour faciliter la communication avec userspaces vu que c'est en vogue ?

VI. Conclusion

Through this work we presented SaaKM, a kernel framework to write schedulers as Linux kernel Modules. Its goal is to provide an easier way than the scheduling class without the need to learn a new language or total subsystem. We show that SaaKM is capable of similar performances than the ext scheduling class with a minimal per-cpu FIFO scheduler.

note:*[idees que je ne sais pas trop où mettre pour le moment : utiliser les modules en C c'est bien car ça forcer les dev à apprendre une nouvelle API ou un langage ça peut être une barrière et les rebuter.*
la perte de sécurité comparé au Enoki/ext n'est pas si grave car si dev expérimenté, il a quand même les outils pour savoir ce qu'il fait et tester son code avec les outils existants.
chaine de compilaton beaucoup plus simple qu'Enoki(besoin de compiler du rust (j'ai pas regardé encore comment compiler leurs sched) et que sched ext)]

References

[1] J. Corbet. (2022, Sep.) Hybrid scheduling gets more complicated. LWN.net article. [Online]. Available: https://lwn.net/Articles/909611/

[2] ——, "An eevdf cpu scheduler for linux," *LWN.net*, Mar 2023, news article. [Online]. Available: https://lwn.net/Articles/925371/

[3] I. Molnar, "Modular scheduler core and completely fair scheduler (CFS)," *LWN.net*, Apr. 2007, kernel announcement. [Online]. Available: https://lwn.net/Articles/230501/

[4] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis, "ghost: Fast & flexible user-space delegation of linux scheduling," in *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, R. van Renesse and N. Zeldovich, Eds. ACM, 2021, pp. 588–604. [Online]. Available: https://doi.org/10.1145/3477132.3483542

[5] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, J. R. Lorch and M. Yu, Eds. USENIX Association, 2019, pp. 361–378. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[6] Y. Jia, K. Tian, Y. You, Y. Chen, and K. Chen, "Skyloft: A general high-efficient scheduling framework in user space," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*, E. Witchel, C. J. Rossbach, A. C. Arpaci-Dusseau, and K. Keeton, Eds. ACM, 2024, pp. 265–279. [Online]. Available: https://doi.org/10.1145/3694715.3695973

[7] T. Heo, D. Vernet, J. Don, and B. Rhoden, "sched_ext: Extensible scheduler class," Linux Kernel Documentation, October 2024, merged in Linux 6.12. [Online]. Available: https://docs.kernel.org/scheduler/sched-ext.html

[8] eBPF Foundation. (2025) ebpf.io — learn, explore, and collaborate on ebpf. Official community portal for the eBPF technology and ecosystem. [Online]. Available: https://ebpf.io/

[9] S. Miller, A. Kumar, T. Vakharia, A. Chen, D. Zhuo, and T. E. Anderson, "Enoki: High velocity linux kernel scheduler development," in *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 2024, pp. 962–980. [Online]. Available: https://doi.org/10.1145/3627703.3629569

[10] Linux Kernel Organization, *KGDB: Linux Kernel Debugger*, 2025, accessed: 2025-08-26. [Online]. Available: https://www.kernel.org/doc/html/v6.12/dev-tools/kgdb.html

[11] M. Larabel and M. Tippett, "Phoronix test suite," benchmarking software. [Online]. Available: https://www.phoronix-test-suite.com/

[12] "Hackbench," benchmarking tool for Linux kernel scheduler (part of rt-tests). [Online]. Available: https://manpages.debian.org/trixie/rt-tests/hackbench.8.en.html

[13] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive scheduling for μsecond-scale tail latency," in *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, J. R. Lorch and M. Yu, Eds. USENIX Association, 2019, pp. 345–360. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/kaffes

Table II
BENCHMARKS RESULTS. THE GAIN/LOSS COLUMN REPRESENTS THE DIFFERENCE BETWEEN EXT AND SAAKM IN PERCENTAGE.

Application - Metric	Reading	ext	SaaKM	Gain/Loss
blender - Time (s)	↓	141.99 ± 0.52	140.25 ± 0.33	1.24%
Linux Kernel Compilation - Time (s)	↓	214.66 (± 2.63)	215.31 (± 2.17)	0.30
cassandra - Operations per second (ops/s)	↑	102686.36 ± 499.01	103036.42 ± 593.6	0.34%
clickhouse_third-run - Queries per minute	↑	126.23 ± 1.39	123.43 ± 1.42	-2.24%
cloverleaf - Time (s)	↓	152.81 (± 0.34)	153.09 (± 0.45)	0.18
7zip Compression - MIPS	↑	58262.14 (± 174.44)	57350.00 (± 98.97)	-1.58
7zip Decompression - MIPS	↑	71097.98 (± 69.42)	71372.50 (± 34.62)	0.39
dav1d - fps	↑	188.02 (± 0.34)	188.64 (± 0.32)	0.33
ffmpeg - fps	↑	211.36 (± 0.52)	211.49 (± 0.40)	0.06
hackbench - Time (s)	↓	33.3 ± 0.03	32.75 ± 0.03	1.68%
namd - ns/day	↑	0.64 (± 0.00)	0.64 (± 0.00)	0.13
npb - Mop/s	↑	21992.45 ± 589.58	22089.27 ± 599.33	0.44%
rbenchmark - Time (s)	↓	0.57 (± 0.00)	0.57 (± 0.00)	0.22
stockfish - Nodes/s	↑	8392488.30 (± 47463.53)	8427082.62 (± 43999.91)	0.41
svt-av1 - fps	↑	95.09 (± 0.14)	95.13 (± 0.13)	0.04
x265 - fps	↑	57.50 (± 1.21)	58.94 (± 1.32)	2.48