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, crucial for both performance and security. Its significance has grown with the increasing complexity of hardware and the high concurrency of modern applications, which frequently utilize numerous threads [1]. Linux provide a general purpose scheduler, Earliest Eligible Virtual Deadline First (EEVDF) [2] that is designed to handle those constraints. However, due to its genericity, it is not tailored to specific workloads and offers worse performance than application-targeted scheduling algorithms [3], [4].

Writing a Linux scheduler is a highly complex endeavor. Beyond meeting performance requirements, it demands a thorough understanding of kernel APIs and interactions with numerous subsystems. The prevalence of multicore hardware introduces additional challenges, including concurrency, synchronization, and managing diverse execution contexts such as interrupts and preemption. These complexities pose substantial obstacles for developers aiming to modify or implement schedulers. Furthermore, any modification typically necessitates recompiling and redeploying the entire kernel, a process that is time-consuming. Errors in scheduler code can result in kernel crashes or data corruption, making debugging particularly challenging.

To address these challenges, several initiatives have emerged to make scheduler development easier, faster, and safer for kernel developers. Two main approaches exist. The first, exemplified by ghOSt [3] and Skyloft [5], enables users to implement schedulers in userspace by exposing scheduling data and providing communication channels between user and kernel space. The second approach focuses on in-kernel schedulers: sched_ext [6], introduced in Linux v6.12, allows writing schedulers as eBPF [7] programs, while Enoki [8] leverages Linux kernel modules to implement scheduling policies in Rust, which are then dynamically loaded into the kernel.

Existing solutions require developers to learn new languages such as Rust or master subsystems like eBPF, introducing a significant learning curve and altering established kernel development workflows. These approaches often restrict access to kernel APIs and internal data structures, limiting flexibility and integration with existing kernel components. To address these limitations, we propose SaaKM: a new approach that streamlines scheduler development by allowing developers to use familiar tools and workflows, prioritizing accessibility and seamless integration with the kernel rather than introducing new paradigms or languages.

SaaKM abstracts thread management using a well-defined state machine, where scheduler behavior is driven by a set of explicit events. This approach enables a generic API—similar to the VFS abstraction for filesystems—that standardizes interactions between the kernel and scheduling policies. By clearly delineating control flow and access paths, SaaKM simplifies the implementation of new scheduling algorithms and enhances modularity. With SaaKM, new schedulers can be implemented as modules that are dynamically loaded into the kernel. Multiple schedulers can coexist, and processes can be flexibly assigned to different scheduling policies, allowing fine-grained control and optimization for diverse workloads.

In this paper, we first review the current approaches to scheduler development. We then introduce the design and implementation of SaaKM, our novel framework for writing scheduling policies as Linux kernel modules. Finally, we compare SaaKM with sched_ext to assess their performance and usability.

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

II. Motivation and Background

The scheduler is a critical part of an Operating System (OS) that manages processes execution. It has to take into account the hardware it runs on but also have to adapt to a lot of different workloads. Currently, the general purpose scheduler of Linux is Earliest Eligible Virtual Deadline First (EEVDF) that has been introduced in v6.6. Being the default scheduler, it has to support a wide variety of workloads (e.g video encoding, web services, ...) but this genericity comes with a cost. Firstly, il is composed of 7000+ LOC and depends heavily on other kernel APIs like the memory management subsystem, the synchronization mechanisms (e.g locks, RCU, ...). This make the development and maintenance quite complex for the kernel developers as you need to put a lot of time and effort to be able to understand the code. Being a core part of the kernel, modifying it requires you to recompile and redeploy your kernel, making the developemnt and debuging a time consuming process. That's why there are been efforts to provide kernel developers ways to write, test, debug and deploy schedulers easily.

There has been an effort in recent years to provide such a solution. We can devide those in two main categories, the schedulers executing in userspace and the ones executing in kernel space.

A. Kernel schedulers

The most straightforward approach to modifying the scheduler is to patch the kernel source directly. Live patching tools such as kpatch [] enable dynamic updates to a running kernel, though they are limited to function-level changes. Consequently, several solutions have been proposed to facilitate the addition of new schedulers to the kernel.

Scheduling class Linux scheduler subsystem offers an API to write scheduler through its *sched_class* structure. It contains a set of function pointers that has to be implemented by the schedulers and we will be called in the right place by the scheduler core. You need to modifiy some scheduler related datastructures to store data of your scheduler. This already represents a lot of work and time because you need to understand how the core of the scheduler (more than 6000 LOC) works. It heavily relies on complex synchronization mecanisms such as locks, Read-Copy-Update (RCU) and bariers due to the inherent concurrent nature of a multicore architecture. Once the scheduling class is implemented, you need to modifu the vmlinux.lds.h linking file to add your scheduling class. This allows the kernel to load and register it in its list, as shown in Fig. **??** showing all of the current scheduling classes in the Linux knerel. You then need to modify the scheduler init function to modify some checks that ensures that all scheduling classes are loaded properly. Finally, you can compile and boot your kernel. Only then, you will be able to test and debug it. Writing a scheduler with the scheduling class API is a long and tedious process that requires a wide and deep knowledge of the Linux kernel that can be a barrier for developers.

Plugsched - Live scheduler update Plugsched [9] takes another approach to give users a way to tweak and modify the scheduler. It leverage the modularization of the Linux kernel by isolating the scheduler code into a *Linux kernel module* that users can then modify, recompile and insert it into a running kernel. To do so, they rewrite the code sections of threads to redirect modified functions. Once you have modified the extracted code, you can insert it. They do a data update during the insertion to migrate the data from previous scheduler version to the new one.**note:***[reformuler, on est forcés de de passer tous les threads dans la nouvelle politique]*

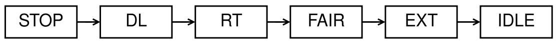

Figure 1. Linux scheduling classes overview

Enoki - Rust schedulers Enoki [8] is a framework that allows users to write scheduling policies as Rust programs. They propose an interface compiled into the kernel, Enoki-C, that manages the schedulers. The second part is libenoki, a library written in Rust and compiled into a Linux kernel Module with the rust written schedulers. Their schedulers API imitate the scheduling one with some changes, such as isolated functions for specific cases (e.g. task_woken). The interface relies on messages passing. They leverage the safety of Rust (there are still some necessary parts of unsage Rust code) to provide security. They also provide a live update mecanism that allows to upgrade a running scheduler without having to unload the previous one. Finally they implemented a Record and Replay mecanism that can be used to record scheduling events and help debugging.

sched_ext - eBPF schedulers The ext scheduling [6] class is a proposition by Meta that has been merged into the Linux kernel mainline in version v6.12. It offers a way to write schedulers as *eBPF* programs. Their goal is to be capable of quickly test and tweak schedulers. The eBPF verifier provides securtiy and safety as it ensures that the program is correct before loading it. It can also be an issue because of the limitations it imposes, policies must heavily rely on helper functions that they define**note:***[retrouver la ref dans enoki je crois?]*. The eBPF API allows to communicate with userspace program through maps, making it possible to delegate scheduling decisions to userspace. For instance, in their scheduler *scx_rusty*, where they offload all of the load balancing logic to a userspace program written in Rust.

B. Userspace schedulers

Numerous user-level threading libraries exist, especially for high-performance computing (HPC) and concurrent applications. However, these solutions are limited to intra-application scheduling and do not provide coordination or control across multiple applications or system-wide resource usage. As such, they fall outside the scope of this paper.

However, some solutions aim to externalize the scheduler to userspace, similar to the microkernel approach or the way FUSE enables userspace filesystems. This allows scheduling decisions to be made outside the kernel while still managing all system processes.

ghOSt - Userpace agents Google proposed ghOSt [3] is an infrastructure that delegates scheduling decisions to userspace. It is composed of two parts. The first one lies in the kernel and implement a scheduling class that interfaces itself with the core scheduler. The second part is executed in userspace and is made of agents. Agents implement scheduling policices thanks to a rich API exposed by the kernel part. The scheduling class exposes scheduling information through messages queues then they inform the kernel part of ghOSt of their scheduling decisions by commiting transactions via shared memory.

Skyloft - Userspace interrupts (impossible de trouver le code alors qu'il y a des artifacts ?) Skyloft takes yet another approach at userspace scheduling. They isolate CPUs core to be exclusively used by their userspace scheduler expect for some background kernel threads. They create kernel threads per isolated cores that are bound to these and a kernel module exposes an ioctl interface for userpsace program to communicate its scheduling decisions. They achieve userspace us-scale scheduling preemption by leveraging usqerspace interrupts (UINT), thus bypassing the kernel. It allows interrupts to be delivred to userspace without swapping adress space or privilege level.

C. Conclusion

todo:*[In summary, the proliferation of new scheduler frameworks and APIs over the past five years—driven by major industry players—clearly demonstrates the pressing need for simpler and faster ways to implement custom scheduling policies. While these solutions have significantly lowered the barrier to experimentation and deployment, they often come with trade-offs in terms of performance, security, or flexibility. The challenge remains to design approaches that combine ease of development with robust integration and minimal overhead, ensuring that both kernel and userspace schedulers can be tailored to diverse workloads without compromising system reliability.]*

III. Design

In this section we present the design of SaaKM, our framework for writing schedulers as Linux kernel modules. First we present the goals of our approach, then we detail the design architeture, and finally we go through the implementation details.

A. Design goals

Writing schedulers in Linux is a tedious task so one of our main goals is to provide a simpler way to do that. The need to have a deep and wide knowledge of the Linux kernel can discourage users. We aim to hide as much complexity as possible. Finally, testing and debugging a scheduler is also a criteria.

Ease of Use An ideal solution would not require kernel developers to master a new langage nor a new subsystem of the kernel as it would defeat one the purpose of the solution, that is the accessibility part. By the nature of kernel development, they must already be knowledgeable in C and some of the kernel APIs and subsystems. Linux kernel modules answer perfectly to this need. Using modules also allows for a light chain of compilation wihtout relying on external tools.

Performance The solution must incur the lowest overhead possible. As the scheduler is a critical part regarding performances, even a small overhead can cause significant performances degradation. We must keep our solution as ligthweiht as possible in order to not loss perforamnces**note:***[reformuler]*

Testing and Debugging The current process of debuging and testing a scheduler class require to recompile and redeploy your kernel image each time you make a modification. This can be quite time consuming and slow down the development process. Our solution must allow users to deploy, test and debug their schedulers quickly without the need to reboot the target.

B. Design overview

SaaKM is an API to write schedulers as Linux kernel modules. It exposes a minimal set of exported functions to the modules that can be used to allocate, manage and destroy scheduling policies and data. Fig. **??** shows how SaAKM integrates itself with the current scheduling classes. We position ourselves right before the idle class to not disrupt higher priority classes that other threads runs.

All of the SaaKM design relies on a structure composed of handlers that each policy must implement. Each handler maps to a scheduling event. Events are devided into two categories. Thread events correspond to all scheduling events related to a thread (e.g threak is created, waking up, ...). On the other hand, core events are related to the CPU cores (e.g scheduling tick, core becoming idle, ...). This distinction makes it clear for the user to know what it should do, hiding the complexity of the scheduling class API where a single function can be called from multiple paths. For instance, the enqueue_task function of the scheduling class can be called from numerous places and for different reasons. It can be for a newly forked task, a migrating task, a waking up task. Our goal is to isolated those cases to let the user know what is actually happening without having to how it got called. Table I shows an excerpt of those events.

Furthermore, this division allows us to hide the complexity of the syucrhonization mecanisms of the core scheduler. As multicore and NUMA architectures are quite common now, the scheduler must syncrhonize data across cores quite often as two core may access the same data concurrently. It protects these data through the usage of locks, RCU and low level synchronization mecanisms like memory barriers. Thanks to our design, the user does not need to worry about these and assume that it has the right locks on the data whenever it is needed.**note:***[réecrire cette partie]*

To increase the flexibility of our solution, we support to have multiple policies loaded at the same time. This allows applications to chose the scheduler that is tailored to their needs.

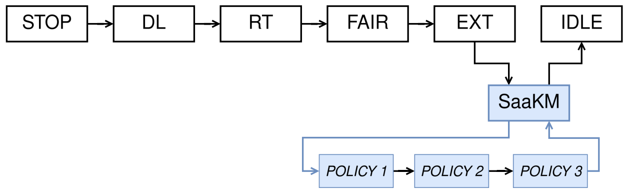

Figure 2. SaaKM scheduling class with three policies loaded (refaire pour rendre plus visible)

C. Implementation

We implement SaaKM on Linux v6.12, the latest longterm release. In this section we will go through the implementation details.

To respect the current Linux scheduler design and ensure compatibility with existing schedulers in the kernel, we implement a minimal (less than 1500 LoC) scheduling class that implements all of the scheduling functions required by the core scheduler. We place ourselves between the ext and idle scheduling classes as shown in Figure 2 where there are three policies loaded. We define a structure `saakm_module_routines` that is composed of all the handler a policy must implement. Table I shows an excerpt of these handlers.

Each policy must allocate and populate a structure with its own handlers and they will be called by our scheduling class at the right places. Each handlers has at least one argument that is a pointer to its corresponding policy structure. Then depending on the event type of the handler (thread or core), it will have either a pointer to a structure `process_event` or `core_event` that contains the data related to the event.

Runqueues management We provide a runqueue implementation that policies uses and includes in their own runqueues as a structure field. This allows to delegate the insertion/deletion and mandatory runqueue metadata fields to be handled by SaaKM. We provide three runqueue types. A FIFO, LIST and RED-BLACK tree. You must provide an ordering function for each type of runqueue, except for the FIFO that will always insert the task at the end of the queue. We provide these three types as they are the most common ones but thanks to its design it can be extended. Policies are required to initialize their runqueues before registering themselves.

Thread states We define our own thread states to make to not interfer with the core scheduler. We define 7 states represented in Fig. 3. We have one special state, SAAKM_READY_TICK that allows policy to notify SaaKM that it needs to trigger a reschedule. With this states, we are able to check for wrongful transitions and provide debug feedback to the user at runtime. This ease the debugging and testing process. We provide a function `change_state(task, next_state, next_cpu, next_rq)` that is the core of SaaKM threads states managmenent Policies must call it anytime they want to alter the state of a thread giving it the right arguments. It is responsible of threads states management. It calls the core schedulers functions to keep the kernel related data correct. It also add and remove threads from the runqueues and check that the operation is valid (i.e. the correct locks are held). Threads states transitions checks is done in this function.

Thread Migration To handle thread migration that occurs because of affinity change or a `exec` rebalance (a newly created thread is being run on a diffrent CPU than its parent), we introduce a new flag, `OUSTED`. This flags allows us to know that we are being dequeued becuase of those reason so we can simulate a fast block/unblock sequence to remove the thread from its current runqueue and insert it into the new one.

Hardware topology To allow policies to properly make load balancing, we export a per-cpu variable, `topology_levels` that represent the underlying hardware toplogy. To do so, we rely on the already existing scheduling domains that are build at initialization of the scheduler subsystem. Policy can then take educated decisions based on the underlying hardware.

Load balancing A central and critical part of scheduling is the load balancing. It consist of trying to distribute the load accross CPU cores according to some metrics. To do so, you may need to access data belonging to other cores. Locks are therefore needed to ensure the correctness of the data. We identified two types of load balancing. Periodic load balancing occurs at a fixed time interval and idle balancing. We define a new software interrupt, `SCHED_SOFTIRQ_IPANEMA` with a handler that calls policies `balancing_select` function. Policies can them chose at which time interval they want to do load balancing. We also raise our interrupt when the fair scheduling kicks a CPU to do the `NOHZ` balancing. Policies should also be able to do load balacing when a CPU is becoming idle. Whenever that happens, we call the policy handler to let it perform it.

Multi-policy support SaaKM supports to have multiple policies loaded at the same time. When it registers, it is given an unique incremental id that is used by application to select their policy. We then maintain them in a linked list ordered by insertion order. When a new thread must be elected, we go through the list of policies starting from the head of the list. The first policy to return a non NULL thread stops the search and the thread is elected. For load balancing, we also go through the full list to make sure that all policies can perform their load balance.

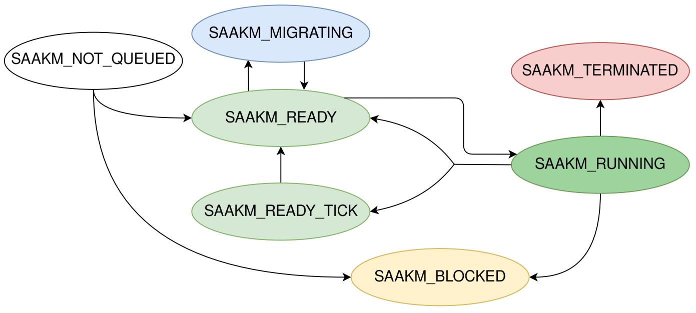

Figure 3. SaaKM states finite state machine

D. SaaKM workflow

In this section we will go through SaaKM workflow ....

Registering a policy To register a policy, during its initialization, a module must allocate and fill a `saakm_policy` that contains all of the policy metadata (i.e. name, routines, id and metadata). It must also allocate then initialize its runqueues with the `saakm_runqueue_init` specifying the type of the runqueue (FIFO, LIST, RED-BLACK TREE) and an ordering function if needed. It can then register itself by calling the `saakm_add_policy`.

Selecting a policy Selecting a policy is pretty straighforward. An application must define an `sched_attr` structure and set the `sched_policy` field to SCHED_SAAKM and set the `sched_saakm_policy` with the value of the policy id it wants. From this point, the thread `task_struct.sched_class` field is set to our scheduling policy and will therefore be scheduled by SaaKM.

Debugging and testing Being a kernel module, policies can leverage the already existing kernel debugging tools such KGDB [10], ftrace, the debugfs interface and many others. On top of that, we provide a kernel configuration options to check the transitions of the thread states and trigger panic when it happens to give user control and a way to inspect the kernel. We also exposes a procfs directory that list the current policies loaded with some metadata. Finally, we use the `sysfs` interface to turn on or off some debugging features : thread transitions checks, print in dmesg the wrongful transitions and print the poilcy thread transitions checks, print the wrongful transitions and print the scheduling classes calls.

Table I
EXTRACT OF `SAAKM_MODULE_ROUTINES` FUNCTIONS

Function	Description
Thread events	
new_prepare(p)	Return CPU id where `p` should run
new_place(task, core)	Insert `p` into `core` runqueue
new_end(task, core)	Insert `p` into `core` runqueue
unblock_place(task)	Return CPU id where `p` should run
unblock_prepare(task)	Insert `p` into `core` runqueue
block(task)	`p` is blocking
yield(task)	`p` is yielding the CPU
tick(task)	Called at each scheduler tick
terminate(task)	`task` dies
Core events	
schedule(core)	Called when a task must be elected
newly_idle(core)	Called right after `schedule` if no READY found
enter_idle(core)	No task to run
exit_idle(core)	A task is runnable
balancing_select(core)	Called to do load balancing
core_entry(core)	`core` is becoming online
core_exit(core)	`core` is going offline
Policy management	
init()	Called to initialize policy
free_metadata()	Free policy metadata
checkparam_attr(sched_attr)	Check validity of sched_attr
setparam_attr(sched_attr)	Use sched_attr to set values
setparam_attr(sched_attr)	Use sched_attr to set values

IV. Evaluation

In this section we evaluate SaaKM ease of use and performance. We presesnt a minimal FIFO scheduler implemented in SaaKM and with the the ext scheduler [6] that has been merged in the linux kernel v6.12. Then we compare how both implementations perform on a set of benchmarks. **note:***[Dans quelle partie mettre ce paragraphe et faut-il le mettre ou l'amener autrement ? – We chose to compare oursevles to the ext scheduler because it recently has been merged and the others solutions did not fit our needs. Enoki [8] works on Linux v5.11, which is 21 release old and 1224 commits behind (for the directory kernel/shed/) the v6.12. Breaking changes also happened to the scheduler, notably the EEVDF merge and ... We want to compare ourselves to kernel scheduler framework and not userspace because it answer different needs, that is why we do not compare SaaKM to ghOSt (which works on a kernel v5.11).]*

A. Experimental Setup

Benchmark platform We run our benchmarks on a server running Debien 12 bookworm with a our patched Linux kernel v6.12. The server is equiped with 10 core and 20 SMT threads (Intel(R) Core(TM) i9-10900K CPU @ 3.70GHz), 20MB of L3 cache and 64GO of RAM.

Benchmarks We used the Phoronix test suite [11] software to run our benchmarks and the hackbench [12] scheduler benchmark. We run each applications 50 times and clean the kernel caches between each run. We present the mean and the standard deviation of our results.

Evaluated schedulers In order to compare ourselves to existing solutions, we implement a FIFO scheduler using SaaKM and sched_ext. The goal is to have both take the same scheduling decisions to see if our solution add or remove overhead. Each policy has a per-cpu local FIFO runqueue. Threads are always enqueued on the same CPU based on their PID. We do not perform any load balancing to simplify the design and to have more consistent results and have a more deterministic behabior.**note:***[pas sûr de ça]* Only the threads of the benchmarks are switched ot the 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

Our evaluation shows that SaaKM is a viable solution to write schedulers but it needs deeper tests. In this work we only test a simple per-cpu FIFO scheduler, we need to implement more schedulers such as Shinjuku [13], EEVDF [2] to see how we perform against them.

We did not leverage the multi-policy support of SaaKM, it would be interresting to see how it impacts multiple applications workloads and if the architecture adds overhead due to the policies iteration.

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] 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

[4] 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

[5] 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

[6] T. Heo, D. Vernet, J. Don, and B. Rhoden, "sched: Implement bpf extensible scheduler class," *LWN.net*, Jun. 2024, ([PATCHSET v7] finalized; scheduled to be merged in Linux 6.12). [Online]. Available: https://lwn.net/Articles/978911/

[7] 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/

[8] 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

[9] T. Ma, S. Chen, Y. Wu, E. Deng, Z. Song, Q. Chen, and M. Guo, "Efficient scheduler live update for linux kernel with modularization," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds. ACM, 2023, pp. 194–207. [Online]. Available: https://doi.org/10.1145/3582016.3582054

[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)	ext	SaaKM	Gain/Loss
blender	141.96 (± 1.13)	141.83 (± 1.11)	-0.09
build-linux-kernel	214.66 (± 2.63)	215.31 (± 2.17)	0.30
cassandra	102458.62 (± 1448.12)	104679.08 (± 1428.35)	2.14
clickhouse	118.79 (± 3.97)	119.47 (± 3.80)	0.57
cloverleaf	152.81 (± 0.34)	153.09 (± 0.45)	0.18
compress-7zip	58119.40 (± 367.40)	57426.78 (± 275.20)	-1.20
dav1d	188.02 (± 0.34)	188.64 (± 0.32)	0.33
ffmpeg	211.36 (± 0.52)	211.49 (± 0.40)	0.06
hackbench	33.03 (± 0.10)	32.61 (± 0.07)	-1.27
namd	0.64 (± 0.00)	0.64 (± 0.00)	0.13
npb	21716.49 (± 1497.52)	22151.99 (± 1700.25)	1.99
rbenchmark	0.57 (± 0.00)	0.57 (± 0.00)	0.22
stockfish	8392488.30 (± 47463.53)	8427082.62 (± 43999.91)	0.41
svt-av1	95.09 (± 0.14)	95.13 (± 0.13)	0.04
x265 (FPS)	57.50 (± 1.21)	58.94 (± 1.32)	2.48