SchedBPF - Scheduling BPF programs

Kavya Shekar
Virginia Tech
Blacksburg, Virginia
skavya@vt.edu

Dan Williams
Virginia Tech
Blacksburg, Virginia
djwillia@vt.edu

ABSTRACT

The Linux BPF framework enables the execution of verified custom bytecode in the critical path of various Linux kernel routines, allowing for efficient in-kernel extensions. The safety properties and low execution overhead of BPF programs have led to advancements in kernel extension use-cases that can be broadly categorized into tracing, custom kernel policies, and application acceleration. However, BPF is fundamentally event-driven and lacks native support for periodic or continuous tasks such as background tracing, metric aggregation, or kernel housekeeping. Existing approaches such as kernel modules with kthreads, userspace daemons, or BPF timers fail to satisfy all the essential requirements for periodic kernel extensions such as fine-grained CPU control, kernel safety, and minimal overhead.

To address this gap, we propose ***SchedBPF*** — a conceptual framework that enables periodic execution of BPF programs on kernel threads. SchedBPF program executions are sandboxed and preemptible, as governed by the existing BPF verifier and JIT engine. They also adopt time-slice semantics, cgroup-style CPU quotas, and nice-level priority control, similar to kernel threads. SchedBPF aims to enable low-overhead, periodic execution of safe BPF code with fine-grained CPU resource management.

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**;

KEYWORDS

Kernel extensions, Kernel modules, eBPF

ACM Reference Format:
Kavya Shekar and Dan Williams. 2025. SchedBPF - Scheduling BPF programs. In *3rd Workshop on eBPF and Kernel Extensions (eBPF '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3748355.3748366

1 BACKGROUND AND MOTIVATION

The ability to run custom bytecode in the kernel at various hook points has enabled various use cases. Monitoring and tracing applications such as Cilium and Hubble [8] leverage BPF to trace packet flows and export metrics with minimal overhead. Application acceleration use cases such as XDP [1] and XRP [5] short-circuit the kernel's traditional network and storage stacks, respectively, resulting in significant performance gains. Kernel extension frameworks like sched_ext [13] and proposals like eBPF-mm [14] allow developers to customize core kernel policies such as process scheduling and memory management. These advancements in kernel extensibility can be attributed to the following intrinsic features of BPF programs:

- **Kernel Programmability**: BPF allows custom programs to be loaded and executed at kernel hook points, allowing the flow of kernel execution to be programmable without system reboots or kernel recompiles.
- **Verifier-Enforced Safety**: The BPF verifier ensures safe in-kernel execution by statically verifying invalid memory accesses, program termination and absence of harmful recursive tail-calls.
- **Minimal Execution Overhead**: BPF programs run in the context of their triggered hook points with minimal context-switch overheads.

However, there exists another class of kernel extension use cases that require periodic or continuous execution. For example: (1) background tracing and profiling tasks such as DAMON [2] require sampling memory access patterns or system statistics at regular intervals; (2) periodic aggregation and garbage collection of per-CPU metrics or the contents of BPF maps; (3) custom kernel daemon tasks for housekeeping activities, such as Kernel Samepage Merging (KSM) or the Out-of-Memory (OOM) reaper. The ideal requirements for such periodically executed kernel extensions are:

(1) **Fine-grained CPU control:** The execution of kernel extensions must be visible as distinct tasks to the Completely Fair Scheduler (CFS), allowing for cgroup-style CPU quotas and nice-level adjustments to ensure controlled resource usage.
(2) **Kernel safety:** The execution of kernel extensions should not compromise the integrity of the kernel and other applications
(3) **Minimal execution overhead:** The overhead associated with periodically scheduling and executing the extension programs should be minimized to maintain system performance.

Considering kernel modules and BPF programs as the two feasible options for kernel extensions, we have identified three different approaches for the periodic execution of kernel extensions. However, these approaches do not satisfy all the key requirements, as summarized in Table - 1

Kernel Threads (kthreads) + Kernel Module: One method for periodic execution of kernel extension involves creating a dedicated kthread via a kernel module, an approach adopted by DAMON[3]. The execution priority of these kthreads could be managed by attaching them to specific cgroups. The overheads of running kthreads can also be bounded, as achieved by DAMON [6]. However, kernel modules are unrestricted and operate with full kernel

eBPF '25, September 8–11, 2025, Coimbra, Portugal

ACM ISBN 979-8-4007-2084-0/25/09.
https://doi.org/10.1145/3748355.3748366

Table 1: Comparison of existing approaches for periodic execution of kernel extensions

Approach	CPU control	Safety	Overhead
kthread	✓	✗	✓
Userspace daemon	✓	✓	✗
BPF timer	✗	✓	✓
SchedBPF	✓	✓	✓

privileges, which is often undesirable and poses a higher risk for simpler kernel extension tasks.

Periodic BPF Program Execution via User-Space Daemon: The second approach involves triggering a BPF program with the `BPF_PROG_RUN` [10] command periodically from a user-space daemon process, as adopted by XDP developers for garbage collection before the introduction of `bpf_timer`[4]. The BPF verifier ensures kernel safety. Since the user-space daemon acts as an orchestrator for the execution of the BPF program, attaching the daemon to a custom cgroup allows for fine-grained CPU control of the kernel extension. However, this approach adds context-switch overheads from user space to kernel space for every invocation of the BPF program.

BPF Timers: Although primarily designed for event-driven scenarios, `bpf_timer`[4] callbacks can be used as a workaround to achieve periodic scheduling of BPF programs. However, BPF timers typically execute in a softIRQ context which is not directly managed by the CFS scheduler. The softIRQ context makes it difficult to specify fine-grained CPU scheduling controls which adds preemption overheads to other processes and fails to intuitively bring out periodic nature of the program. eTran [11] addresses a few of these limitations through modifications to BPF timers, highlighting both the upcoming need for periodic execution of kernel extensions and the absence of a native BPF support.

In summary, none of the existing approaches comprehensively satisfies all the key requirements for a periodically executed kernel extension. To address this gap, we propose a new framework — SchedBPF — designed to satisfy all the requirements for a periodically executed kernel extension.

2 PROPOSED SOLUTION

SchedBPF is a conceptual framework that introduces a new BPF program type : `BPF_PROG_TYPE_PERIODIC`. During program load time, the BPF program is verified and a dedicated (`kthread`) is created for triggering the BPF program. The periodicity of execution, as specified by the user, is used to configure the schedule interval for the execution of the BPF program. The newly created `kthread` will be assigned to a custom cgroup, whose scheduling attributes will have default values suitable for running periodic BPF programs with minimal overhead and interference to other processes. To summarize, the proposed design addresses the key concerns as follows:

- **Fine-grained CPU control:** Managed through custom cgroup configuration.
- **Kernel Safety:** Achieved via the existing BPF verifier and sandboxing mechanism.
- **Low overhead:** Achieved by using kthreads, enabling direct in-kernel scheduling of programs.

3 DISCUSSION

Controlling Periodicity: Two approaches are proposed for the periodic execution of a BPF program: (1) *Framework-managed*: The `kthread` is periodically woken up to run the BPF program, separating the BPF program's logic from the scheduling mechanism. (2) *BPF Program-managed*: The `kthread` triggers the BPF program once, and the program itself manages its periodicity by sleeping in a non-blocking way after each execution.

Specialized `kthread` Execution: To improve performance, we could explore optimizing the execution of `kthread`. This could involve minimizing the BPF sandbox setup or reducing context switch overhead for waking up the `kthread`, given that the `kthread`'s sole function is to run the BPF program.

Dedicated vs. Shared `kthread`: The concept of a dedicated `kthread` per BPF program is still open to discussion. An alternative is to allow multiple related BPF programs to share a single `kthread`, in the Linux workqueue style. This would enable unified resource management and control for a group of programs.

System-wide Impact and Cross-Core Inference: For profiling, new BPF helper functions could be introduced that gather metrics from multiple cores. This necessitates considerations of cross-core interference from the `kthreads` [9] and the potential use of `cgroups` to manage and mitigate these effects.

Preemption of BPF Programs: Sleepable BPF programs [12] and the introduction of `bpf_preempt_enable()`[7] demonstrate that BPF programs, together with a specific subset of kfuncs marked as sleepable, can safely operate in preemptible contexts. However, a more comprehensive investigation may be necessary to identify potential issues or edge cases related to CFS-driven preemption during BPF program execution.

4 SUMMARY

In this extended abstract, we have motivated the need for periodic execution of kernel extensions in the Linux kernel and identified the shortcomings of existing approaches. SchedBPF offers a conceptual framework for a new type of BPF program that provides safe, preemptible, and resource-controlled periodic execution of BPF programs. We believe that SchedBPF will open the doors to a new set of kernel extension use-cases that are periodic in nature all while also maintaining the benefits of BPF safety and performance.

5 ACKNOWLEDGEMENT

This work was funded in part by NSF grant CNS-2236966

6 ETHICAL CONCERNS

This work raises no ethical concerns

REFERENCES

[1] 2018. The eXpress data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging*

Networking EXperiments and Technologies (CoNEXT '18). Association for Computing Machinery, New York, NY, USA, 54–66. https://doi.org/10.1145/3281411.3281443

[2] 2019. DAMON. https://damonitor.github.io/. (2019).

[3] 2019. DAMON design document. https://www.kernel.org/doc/html/v6.12/mm/damon/design.html. (2019).

[4] 2021. [PATCH v5 bpf-next 00/11] bpf: Introduce BPF timers. (2021). https://lwn.net/ml/bpf/20210708011833.67028-1-alexei.starovoitov@gmail.com/

[5] 2022. XRP: In-Kernel Storage Functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 375–393. https://www.usenix.org/conference/osdi22/presentation/zhong

[6] 2023. DAMON Evaluation. https://damonitor.github.io/posts/damon_evaluation/. (2023).

[7] 2024. [PATCH bpf-next v1 0/2] Introduce bpf_preempt_disable,enable. (2024). https://lore.kernel.org/all/20240423061922.2295517-1-memxor@gmail.com/

[8] 2025. Cilium and Hubble. https://docs.cilium.io/en/stable/overview/intro/. (2025).

[9] 2025. [RFC/PATCH] sched: Support moving kthreads into cpuset cgroups. (2025).

[10] 2025. Running BPF programs from userspace. (2025). https://docs.kernel.org/bpf/bpf_prog_run.html

[11] Zhongjie Chen, Qingkai Meng, ChonLam Lao, Yifan Liu, Fengyuan Ren, Minlan Yu, and Yang Zhou. 2025. eTran: Extensible Kernel Transport with eBPF. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 407–425. https://www.usenix.org/conference/nsdi25/presentation/chen-zhongjie

[12] Jonathan Corbet. 2020. Sleepable BPF programs. (2020). https://lwn.net/Articles/825415/

[13] Jonathan Corbet. 2023. The extensible scheduler class. (2023). https://lwn.net/Articles/922405/

[14] Konstantinos Mores, Stratos Psomadakis, and Georgios Goumas. 2024. eBPF-mm: Userspace-guided memory management in Linux with eBPF. (2024). arXiv:cs.OS/2409.11220 https://arxiv.org/abs/2409.11220