The Impact of Kernel Asynchronous APIs on the Performance of a Kernel VPN

Honore Cesaire Mounah
cesaire.mounah-honore@inria.fr
Univ. Rennes, Inria, CNRS, IRISA
Rennes, France

Djob Mvondo
barbe-thystere.mvondo-djob@inria.fr
Univ. Rennes, Inria, CNRS, IRISA
Rennes, France

Julia Lawall
julia.lawall@inria.fr
Inria
Paris, France

Yerom-David Bromberg
david.bromberg@irisa.fr
Univ. Rennes, Inria, CNRS, IRISA
Rennes, France

Abstract

Linux kernel VPNs suffer from severe performance degradation under high load due to execution order inversion (*EoI*), a phenomenon where packet recombination functions preempt earlier pipeline stages. This leads to severe latency spikes and throughput reductions. We investigate kernel threads and workqueues as alternative kernel asynchronous APIs to address these limitations, achieving up to a 4.7× increase in throughput while reducing tail latency by 65%. These results demonstrate the importance of selecting appropriate kernel asynchronous APIs for kernel-level network applications.

CCS Concepts

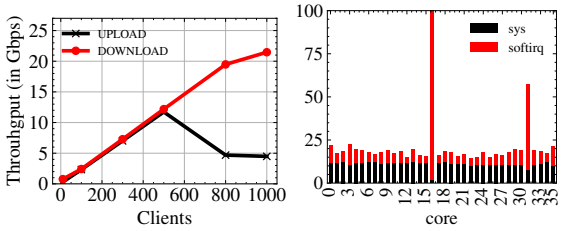
• **General and reference** → **General conference proceedings**; • **Networks** → **Network performance evaluation**; • **Software and its engineering** → **Scheduling**; **Software performance**.

Keywords

Operating System, WireGuard, VPN, Asynchronous Execution, Kernel Threads, Workqueues

1 Introduction

Millions of users rely daily on virtual private networks (VPNs) for privacy and security [13, 16, 17]. WireGuard [4] is a VPN protocol that is gaining significant traction. It is implemented as a Linux kernel module to improve execution speed and is designed to be multithreaded, taking advantage of multicore architectures to handle high loads [3, 4]. In practice, however, we find that, when serving a high number of users, the performance of the kernel WireGuard falters. We consider a scenario where up to 1,000 clients each generate 25 Mbps of upload (received) and download (transmitted)[1] traffic to a WireGuard server equipped with a 25 Gbps NIC and an 18-core CPU. As shown in Fig. 1a, the transmit traffic throughput scales nearly linearly up to 1,000 clients, reaching 22 Gbps. However, the received traffic throughput is only 4.8 Gbps for 800 to 1,000 clients, which amounts to only 19.2% of the NIC capacity.



(a) Throughput **(b) CPU usage with 1,000 clients**

Figure 1: Network throughput and per-core CPU usage of WireGuard when serving up to 1,000 clients, each generating 25 Mbps in upload and in download.

We find that the performance issue in the reception pipeline stems from the use of kernel asynchronous APIs (hereafter referred to as *KAAPI*). In WireGuard, ingress packet processing involves three main steps: (i) decapsulation, (ii) decryption, and (iii) generic receive offload (GRO), which aggregates packet fragments. In Linux, GRO is executed in a high-priority software interrupt context. As a result, it preempts the decryption step in 8% of calls—frequent enough to

[1]Traffic *uploaded*/*downloaded* by the clients is *received*/*transmitted* by the server.

SYSTOR '25, Virtual, Israel

ACM ISBN 979-8-4007-2119-9/25/09
https://doi.org/10.1145/3757347.3759133

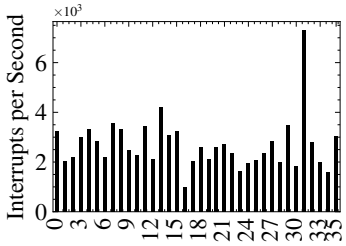


Figure 2: Distribution of the network card interrupts across all CPU cores

severely disrupt packet batching, leading to a mismatch between the logical processing order and the actual execution order.

This mismatch causes what we refer to as execution order inversion (hereafter referred to as *EoI*[2]), where later pipeline stages run before earlier ones have completed. *EoI* has two significant consequences. First, GRO software interrupts are scheduled before any decrypted packets are ready, resulting in immediate aborts and wasted CPU cycles. Second, when decrypted packets are available, GRO processes them in large batches, which takes longer and delays subsequent interrupts. When a core falls behind due to processing a large GRO batch, it accumulates further interrupts, which are serviced later than those on less-busy cores. This allows decrypted packets to pile up unevenly across cores, creating even more work for the overloaded core. Over time, this feedback loop leads to significant load imbalance: busy cores become overwhelmed while others remain underutilized, ultimately degrading overall network throughput (see Fig. 1b).

The distribution of receive (RX) interrupts across cores (Fig. 2) reveals a puzzling discrepancy between interrupt load and CPU saturation. While core 11 receives approximately twice as many RX interrupts as other cores—a moderate imbalance typical of hash-based distribution—this alone cannot explain the severe CPU utilization patterns observed. Crucially, core 18 saturates at 100% softirq utilization (Fig. 1b) despite receiving fewer interrupts than core 11. This counterintuitive observation suggests that RX interrupt distribution, while imbalanced, is not the primary driver of the performance bottleneck. The disproportionate CPU saturation relative to the interrupt count indicates that other factors amplify the initial imbalance into the severe degradation we observe. Consequently, simple interrupt redistribution may not resolve the performance bottleneck.

To address this problem, we propose to make the GRO functions run with the same priority as the other tasks of the pipeline, so that they do not preempt other WireGuard tasks. To this end, we explore execution contexts other than software interrupts that could be used for GRO. In the Linux kernel, there are two *KAAPI* that can achieve this: kernel threads (kthreads) that are already supported in the GRO API and workqueues that are not.

Our results show that for WireGuard's GRO processing, kthreads provide a 4× throughput increase and 65% latency reduction, while workqueues deliver 4.7× higher throughput and 46% lower latency under high-load reception. Transmission performance remains unaffected in both cases. Although kthreads reduce latency more effectively, they achieve lower throughput due to increased scheduling overhead resulting from the use of more threads.

This paper makes the following contributions:

- We show that *EoI* creates load imbalance and severely degrades WireGuard performance.
- We propose a patch to run GRO functions in workqueues, and enable WireGuard to use this feature.
- We improve the throughput of WireGuard by up to 4.7× and the tail latency by up to 50%.

2 Background

We provide an overview of kernel multiprocessing APIs, and the architecture and implementation choices of WireGuard.

2.1 Linux Kernel Asynchronous APIs

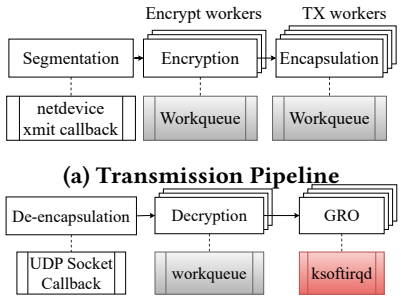
The Linux kernel provides several APIs and mechanisms to enable asynchrony: software interrupts (softirqs), kthreads, and workqueues.

Softirqs. The Linux kernel handles softirqs asynchronously using specialized threads called `ksoftirqd`. Softirqs can be used in networking when a large volume of incoming packets has to be processed, and each packet generates a network interrupt. The NAPI API enables the kernel to mask interrupts for a specified duration and execute interrupt handlers asynchronously. Using NAPI involves defining a `napi_struct` structure coupled with a polling function. `napi_struct` instances are placed in a per-core queue called `softnet_data`. When software interrupts are raised, the `ksoftirqd` threads execute the polling functions of the `napi_struct` in the `softnet_data`.

kthreads. A kthread is a data structure managed by the kernel that can be initialized with a function defining the tasks the thread will perform. A kthread can be created to run the asynchronous code. The kernel scheduler oversees the execution of kthreads. Kthreads are scheduled like any other kernel thread, subject to the scheduler's fairness and load balancing policies.

Workqueues. A workqueue is a Linux kernel mechanism that enables asynchronous task execution through a pool of kthreads called *workers*, executing functions called *works* stored in a *workqueue*. It allows tasks to be distributed across

[2] *EoI* is distinct from "End of Interruption", as these are two different notions.



(a) Transmission Pipeline

(b) Reception Pipeline

Figure 3: Execution contexts of the functions of WireGuard.

multiple CPU cores. Workqueues are analogous to the software interrupt mechanism, with workers corresponding to ksoftirqd threads, workqueues to `softnet_data`, and work items to `napi_struct`.

2.2 WireGuard Internals

WireGuard creates encrypted tunnels between multiple peers. Each WireGuard peer has two pipelines (Fig. 3): one to send packets (Fig. 3a) and one to receive packets (Fig. 3b).

In the transmission pipeline, packets are segmented, encrypted, and encapsulated. Encryption functions run asynchronously in workqueues with one CPU-affine worker thread per core, while encapsulation uses one work item per peer that can be scheduled on any core.

In the reception pipeline, packets are de-encapsulated, decrypted, and recombined. Decryption also uses workqueues for asynchronous processing. After decryption, packets are recombined using the kernel's GRO mechanism, with each peer having its own GRO handler. WireGuard leverages existing kernel APIs for GRO within the NAPI subsystem. GRO functions are thus executed by default in the softirq context.

3 Execution Order Inversion (*EoI*)

This section introduces the *EoI* bottleneck, how it arises, and how it degrades WireGuard performance.

3.1 What is *EoI*?

EoI occurs when the priorities of execution contexts conflict with the intended function execution sequence. Consider a system with a sequence of functions $F_1, F_2, \ldots, F_m$, where each function F_i must execute before F_j, with $i < j$. Each function runs within an execution context G_i, with an associated priority $p(G_i)$. *EoI* arises when, for two functions F_i and F_j with $i < j$, the priority $p(G_i) < p(G_j)$, allowing a later function to preempt an earlier one prematurely. This conflict disrupts the required execution order, introducing inefficiencies.

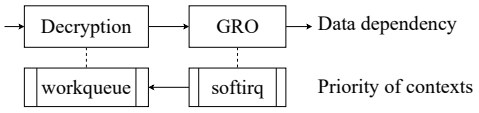


Figure 4: Pipeline Order inversion in WireGuard

3.2 How *EoI* Affects WireGuard

In the reception pipeline of WireGuard (Fig. 3b), decryption functions are dispatched to workqueues with normal priority, while GRO handlers are executed in `ksoftirqd`, which has a higher priority. This priority mismatch results in *EoI* (Fig. 4).

Specifically, when packets are dequeued from encryption/decryption queues, WireGuard uses `spin_lock_bh` and `spin_unlock_bh`. The latter re-enables bottom halves, which immediately trigger the execution of software interrupts such as NAPI pollers. As a result, NAPI pollers may run before decryption has completed, often finding few or no decrypted packets to process, hence preventing efficient batching. The transmission pipeline does not exhibit this issue, as it does not have such mismatches.

3.3 Impact on CPU Utilization

Under high upload workloads, *EoI* results in severe CPU core imbalance. Measurements show one core reaching up to 94% utilization, while others remain around 20% (Fig. 1b).

This happens because most NAPI pollers execute too early, before decryption workers finish, so they return after processing few packets. Meanwhile, some NAPI pollers are delayed by competing threads or pollers for other network devices. These delayed pollers run later and find a backlog of decrypted packets, processing significantly more data. On average, most cores process 1 packet per NAPI invocation. However, the overloaded core processes 5 packets per invocation, a 5× increase in per-invocation runtime.

NAPI pollers are distributed across CPUs without regard to utilization. Specifically, decryption workers are scheduled on different cores in a round-robin manner, independently of the system's current CPU load[3]. When a decryption worker completes its task, it schedules the NAPI poller for the corresponding peer[4]. This is done via a call to `napi_schedule()`, which enqueues the poller on the CPU where the worker is running[5]. Since worker placement does not account for per-core load, heavily loaded cores can continue receiving

[3] `wg_queue_enqueue_per_device_and_peer`: https://elixir.bootlin.com/linux/v6.1.147/source/drivers/net/wireguard/queueing.c#L93

[4] `wg_packet_decrypt_worker`: https://elixir.bootlin.com/linux/v6.1.147/source/drivers/net/wireguard/receive.c#L408

[5] Pollers are scheduled via `wg_queue_enqueue_per_peer_rx`, which calls `napi_schedule`, assigning them to the current CPU via `__napi_schedulehttps://elixir.bootlin.com/linux/v6.1.147/source/net/core/dev.c#L6336`

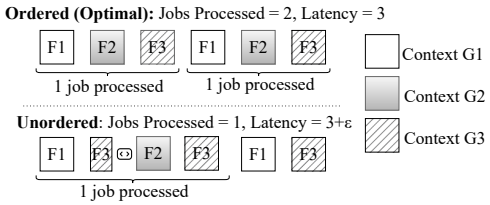


Figure 5: Effect of *EoI* on throughput and latency.

new pollers. In contrast, other cores remain significantly underutilized, resulting in persistent load imbalance, primarily when GRO processing is concentrated on a single CPU.

3.4 Impact on Pipeline Processing Speed

The increased per-invocation runtime of *EoI* also degrades pipeline throughput and increases tail latency. Consider a pipeline with three functions F_1, F_2, and F_3, each taking one unit of time to execute (Fig. 5). In the ideal case, the pipeline processes one job in 3 time units and two jobs in 6 time units. However, with *EoI*, if, for example, F_2 runs before F_3, the system incurs additional delay, and the total time to process one job increases to $3+\epsilon$, where $\epsilon \leq 1$ captures the overhead of disordered execution. Over extended runs, this inversion reduces pipeline throughput (fewer jobs completed per unit time) and increases processing tail latency.

3.5 *EoI* in Other Applications

While our analysis focuses on WireGuard with VPN traffic handling 1,000 distinct client flows, *EoI* represents a broader design consideration for asynchronous pipelined systems. *EoI* can occur in any system where pipeline stages run in different priority contexts. For example, storage stacks or network middleware that mix softirq and workqueue processing may exhibit similar performance degradation.

4 Using Alternative KAAPI in WireGuard

EoI arises because GRO tasks are executed in software interrupt context, which has a higher priority than the execution context of the decryption and de-encapsulation functions. We thus investigate *KAAPI* alternatives to the software interrupt context for GRO tasks. An alternative should (1) eliminate *EoI* by preserving the correct execution order across the pipeline stages, (2) allow high-frequency packet processing, (3) improve upload performance, and (4) maintain good performance on download. For this, we look at *KAAPIs* that have the same priority as the execution context of the decryption and de-encapsulation functions. With the same priority for all pipeline functions, *EoI* should not occur. The alternative *KAAPIs* we consider are kthreads and workqueues, described in Section 2.1. We implement them in the NAPI subsystem, allowing them to be used to execute NAPI pollers. We present the modifications for kthreads and workqueues.

4.1 Running NAPI pollers in kthreads

The NAPI subsystem allows pollers to be executed in kthreads, by writing 1 to `/sys/class/net/<iface_name>/threaded`, where `iface_name` is the name of the network interface whose NAPI pollers should be threaded.

4.2 Running NAPI pollers in workqueues

The NAPI subsystem does not natively support workqueues. To allow this, in the kernel, we encapsulate the `napi_struct` within a `work_struct` and leverage the workqueue infrastructure to decouple NAPI processing from the *KAAPI* supported by NAPI. These changes impact the NAPI initialization step and the NAPI scheduling step.

Initialization. The NAPI initialization step must bridge the gap between the `napi_struct` structure used by NAPI and the `work_struct` structure used by the workqueues. To this end, we replace the original `netif_napi_add` initialization function with a new function `netif_napi_add_wq` and wrap the `napi_struct` in a `work_struct`. `netif_napi_add_wq` then establishes the associations between the `napi_struct`, the `work_struct`, and the workqueue. During the device setup phase, `dev_run_in_workqueue` is called to enable the workqueue mode for every attached NAPI poller.

Scheduling. When `napi_schedule` is called, and the NAPI instance is in workqueue mode, instead of enqueuing the `napi_struct` into the softnet data, the `queue_work_on` function is invoked. This function queues the associated `work_struct` onto the workqueue assigned to the relevant CPU core. The kernel's workqueue subsystem then takes over, executing these tasks using a shared pool of worker threads. These changes add 136 lines of code (LoCs) to the NAPI subsystem and 55 LoCs to WireGuard. These changes do not modify the Linux kernel's behavior in the sense that they only enable NAPI pollers to run in other contexts. So, applications using the default execution context of NAPI handlers are not impacted.

4.3 Kthread vs Workqueues

kthreads are easy to deploy since they require no code modifications and can be enabled by writing to a configuration file (Section 4.1). However, this approach may create scalability challenges. The one-thread-per-client model means that supporting 1,000 clients requires managing more than 1,000 individual threads. This design leads to excessive context switching and frequent thread migrations between CPU cores, which consumes substantial processing resources. To

address the potential scheduling overhead from thread migrations, we also evaluate a version of kthreads where each thread is pinned to a specific CPU core.

Workqueues are more complex to deploy because they require modifications to both the kernel and WireGuard. However, they offer a different architectural approach through their fixed-thread design. They maintain one worker thread per CPU core regardless of client count. As the number of clients increases, the number of work items in the queues also increases, while the thread count remains constant. This design reduces scheduler overhead and avoids the costly context switches associated with the kernel thread approach.

4.4 Impact on the System

In both cases, whether using kthreads or workqueues, only the NAPI pollers of WireGuard are affected. GRO pollers initialized by other drivers will continue to run in the `ksoftirqd` context. As a result, our approach, which executes WireGuard's GRO pollers in a separate context, does not interfere with those of other drivers. Using kthreads can introduce system overhead. Specifically, one kthread is created per peer when a VPN interface is brought up. On servers with many clients, this can lead to a large number of threads, potentially overloading the scheduler and degrading the performance of other applications. In contrast, workqueues are more lightweight. They rely on a fixed number of kernel threads, thereby avoiding the scalability issues associated with kthreads and imposing less strain on the scheduling subsystem.

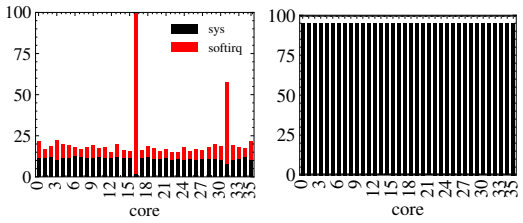
5 Evaluation

The evaluation addresses the following questions: (1) What is the performance of WireGuard with kthreads and workqueues, and how does it compare with the current implementation of WireGuard? (2) Which is better, workqueues or kthreads, and why? (3) Do the solutions also impact transmission?

5.1 Experimental Setup

We describe the test setup and the different metrics used.

Hardware and Software. We use 21 identical machines with an Intel Xeon Gold 5220 CPU with 18 physical cores, with 10 machines for clients, 10 machines as targets, and one machine as the VPN server, all interconnected on the same switch [8]. Each machine has a 25 *Gbps* Mellanox ConnectX-4 NIC, with RSS [2, 9, 14] enabled by default.The full-duplex mode is activated on the NIC. The NIC driver is `mlx5_core` version 5.0. All machines run on Debian 12 using Linux kernel version 6.1. Since WireGuard is integrated directly into the kernel, the WireGuard version corresponds to what is included in Linux kernel v6.1. We use: (1) `iPerf3` (v3.9) [5] to generate network traffic, and (2) `netperf` to evaluate the network latency.



(a) Baseline (softirq) **(b) kthreads/workqueues**

Figure 6: Per core reception CPU usage on 1,000 clients

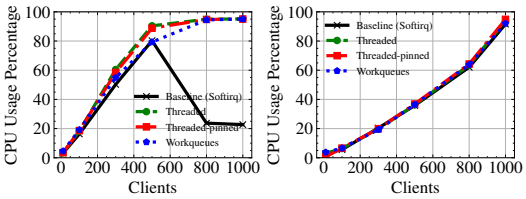
Metrics. We measure the median value of network throughput on the server using `sysstat` [6] by capturing the forwarded throughput on the network interface in Gbps over 60 seconds. Latency is measured with the `TCP_RR` benchmark from `netperf`, and we report the $99th$ percentile latency. Total and per-core CPU usage are measured using `sysstat`. We run each experiment five times.

5.2 Improving the CPU Usage

To evaluate the impact of kthreads and workqueues on the server CPU usage, we analyze the overall CPU utilization as a function of the forwarded throughput and the load distribution across the server's cores for 1,000 clients' traffic.

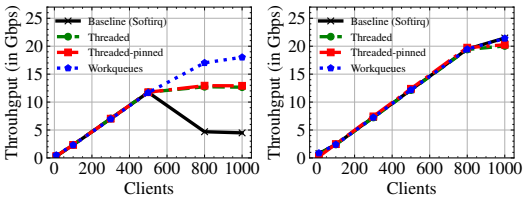
Per-core CPU usage. Fig. 6 shows the load per core when the server manages 1,000 clients in reception. Fig. 6a shows the load for the baseline version. The load is identical for both kthread and workqueue implementations (Fig. 6b). When receiving traffic from 1,000 clients, the baseline shows poor CPU load distribution, with one core at 94% (essentially for softirq) and others at 20%. In contrast, both the kthread and workqueue versions achieve better balance, with up to 95% load across all cores. As GRO is no longer run in softirq, the corresponding CPU usage is only accounted for as kernel CPU time. For transmission, we observe an even distribution for all configurations, similar to the workqueue upload case.

Total CPU usage. Fig. 7 compares the total CPU usage of the WireGuard server when serving different numbers of clients using four implementations: the baseline using softirqs, workqueues, kthreads, and kthreads pinned to specific cores (threaded-pinned). Fig. 7a and 7b present results for scenarios where the server receives and transmits network traffic, respectively. Lower CPU usage and higher throughput indicate better performance, with the optimal configuration represented by the points at the bottom right. For reception, the CPU usage of the threaded versions is higher than that of the workqueue version for the same throughput. Thus, kthread-based configurations require more CPU resources to achieve comparable performance. Using workqueues results in better CPU efficiency than using kthreads or the



(a) Reception **(b) Transmission**

Figure 7: Total CPU usage with up to 1,000 clients



(a) Reception **(b) Transmission**

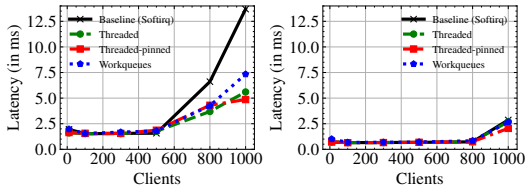
Figure 8: Forwarded throughput for varying number of clients.

baseline. For the transmission, while all variants show linear increases in CPU utilization, the workqueue and baseline configurations achieve the same results. However, kthreads use more CPU for less throughput. Overall, the workqueue configuration is the most efficient, balancing CPU usage and throughput, especially under high loads.

5.3 Network Performance

Network Throughput. Fig. 8 presents the throughput of WireGuard when serving up to 1,000 clients in reception and transmission. For reception, the baseline throughput increases linearly to 500 clients, peaking at 11.7 Gbps, but declines sharply to 4.5 Gbps at 1,000 clients. kthread versions maintain throughput beyond 500 clients, reaching 14.4 Gbps, outperforming the baseline by up to 1.8×; pinning threads gives no advantage. The workqueue version reaches 18 Gbps at 1,000 clients, thus a 4× improvement. For transmission, at 1,000 clients, the baseline and workqueue maintain 21.5 Gbps, while kthreads' throughput drops slightly to 20.2 Gbps.

Network Latency Fig. 9 presents the latency of the WireGuard clients when using four implementations of the WireGuard server. Fig. 9a and 9b present results for scenarios where clients primarily upload (the server receives the traffic) and download (the server transmits the traffic), respectively. The baseline shows a tail latency of 13.7 ms when clients perform upload. kthreads significantly improve this latency, dropping to 5.6 ms without pinning and further to 4.8 ms with pinning (65% improvement). The workqueue improves latency to 7.3 ms (46.7% improvement). For client download, latency remains similar across all variants, increasing slightly



(a) Client upload **(b) Client download**

Figure 9: Tail (99*th*) latency for varying number of clients.

up to 500 clients (0.6 to 0.7 ms) and rising more noticeably for larger loads: 2.8 ms for baseline, 2.7 ms for kthreads and workqueues. The best result is achieved with pinned threads, 2 ms.

6 Other Kernel: WireGuard in FreeBSD

We also evaluate WireGuard on FreeBSD v14.2 using a setup similar to Section 5, with RSS enabled. Like Linux, FreeBSD WireGuard uses two pipelines, but all functions run in *task queue groups*, a *KAAPI* similar to Linux workqueues with uniform priority. As a result, *EoI* does not occur. In server reception, WireGuard reaches 14.5 Gbps for 1,000 clients, matching the performance of threaded NAPI on Linux. Latency is also comparable. However, for server transmission, performance is lower than Linux WireGuard, with 13.9 Gbps throughput and 13.1 ms latency. Using `DTrace` [7], we attribute this to lock contention: task queue group locking accounts for 36.9% of total CPU usage. While outside the scope of this work, this is a potential area for future improvement.

7 Related Work

EoI shares similarities with two known issues. *Receive livelock* [1, 10, 11] occurs when software interrupts consume excessive CPU resources, preventing other tasks from executing and degrading network application performance. *EoI* differs in that software interrupts do not monopolize the CPU; instead, they impact applications through unmanaged dependencies between operations running in ksoftirqd and other applications that must complete before these operations can proceed. *Priority inversion* [12, 15, 18] represents another related but distinct problem. It happens when a lower-priority process holds a resource required by a higher-priority process, effectively blocking the more critical task. While *EoI* might appear similar, the key difference is that in *EoI*, the higher-priority task (GRO in WireGuard) is not stalled because a lower-priority task (like decryption) monopolizes resources. Instead, the higher-priority task often preempts the lower-priority task before completion.

8 Conclusion

Our study shows that *EoI* limits WireGuard's scalability, increasing latency, reducing throughput, and causing CPU imbalance under load. Replacing softirqs with kernel threads helps, but workqueues yield better performance overall. The study underscores the importance of choosing the right *KAAPI* for kernel-level applications.

9 Acknowledgments

We thank our shepherd, Nadav Amit, and the anonymous reviewers for their insightful feedback. This work was supported in part by Inria Défi OS, the ANR projects sGOV (ANR-23-CE25-0007-01) and Seconde Chance. Experiments were conducted on Grid'5000, supported by Inria, CNRS, RENATER, and partner institutions (https://www.grid5000.fr).

References

[1] X. Chang, J. K. Muppala, Z. Han, and J. Liu. 2008. Analysis of interrupt coalescing schemes for receive-livelock problem in gigabit ethernet network hosts. In *2008 IEEE International Conference on Communications*. 1835–1839. doi:10.1109/ICC.2008.352

[2] Linux Kernel Documentation. 2025. *Scaling in the Linux Networking Stack*. Retrieved July 24, 2025 from https://docs.kernel.org/networking/scaling.html#rss-receive-side-scaling

[3] Jason A Donenfeld. 2017. WireGuard Linux Kernel Integration Techniques. *Proceedings of Netdev* 2 (2017), 26–43.

[4] Jason A Donenfeld. 2017. Wireguard: next generation kernel network tunnel. In *NDSS*. 1–12.

[5] Jon Dugan, Seth Elliott, Bruce Mah, Jeff Poskanzer, and Kaustubh Prabhu. 2025. *iPerf3 - The TCP, UDP and SCTP network bandwidth measurement tool*. Retrieved July 24, 2025 from https://iperf.fr/

[6] Sebastien Godard. 2025. *sysstat - System performance tools for the Linux operating system*. Retrieved July 24, 2025 from https://sysstat.github.io/

[7] Brendan Gregg and Jim Mauro. 2011. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional.

[8] Dell Inc. 2023. *Dell Z9264F-ON Switch Doctumentation*. https://www.dell.com/support/product-details/fr-fr/product/networking-z9264f-on/docs Retrieved on July 24, 2025.

[9] Intel. 2025. *Receive Side Scaling on Intel® Network Adapters*. Retrieved July 24, 2025 from https://www.intel.com/content/www/us/en/support/articles/000006703/network-and-i-o/ethernet-products.html

[10] Alex Klinkhamer and Ali Ebnenasir. 2019. On the verification of livelock-freedom and self-stabilization on parameterized rings. *ACM Trans. Comput. Logic* 20, 3, Article 16 (June 2019), 36 pages. doi:10.1145/3326456

[11] Jeffrey C Mogul and Kadangode K Ramakrishnan. 1997. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems* 15, 3 (1997), 217–252.

[12] T. Nakajima, T. Kitayama, H. Arakawa, and H. Tokuda. 1993. Integrated management of priority inversion in Real-Time Mach. In *1993 Proceedings Real-Time Systems Symposium*. 120–130. doi:10.1109/REAL.1993.393508

[13] NordVPN. 2025. Press Area. https://nordvpn.com/press-area/.

[14] Redhat. 2025. *8.6. Receive-Side Scaling (RSS) Red Hat Enterprise Linux 6 |Red Hat Customer Portal*. Retrieved July 24, 2025 from https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-rss

[15] H. Tokuda, C.W. Mercer, Y. Ishikawa, and T.E. Marchok. 1989. Priority inversions in real-time communication . In *1989 Real-Time Systems Symposium*. IEEE Computer Society, Los Alamitos, CA, USA. doi:10.1109/REAL.1989.63587

[16] CyberGhost VPN. 2025. *Why CyberGhost VPN*. Retrieved July 24, 2025 from https://www.cyberghostvpn.com/features/why-cyberghost-vpn

[17] Proton VPN. 2025. 2025 Survey Results. https://proton.me/blog/2025-proton-survey-results.

[18] Yun Wang, E. Anceaume, F. Brasileiro, F. Greve, and M. Hurfin. 2002. Solving the group priority inversion problem in a timed asynchronous system. *IEEE Trans. Comput.* 51, 8 (2002), 900–915. doi:10.1109/TC.2002.1024738