IAPS: Decreasing Software-Based Packet Steering Overhead Through Interrupt Reduction

Maike Helbig
College of Computing and Informatics
Sungkyunkwan University
Suwon, South Korea
hema@g.skku.edu

Younghoon Kim
College of Computing and Informatics
Sungkyunkwan University
Suwon, South Korea
kyhoon@gmail.com

***Abstract*—Packet Steering is often performed in hardware, rendering the software-based packet steering mechanisms of the Linux kernel obsolete. However, the ability to redistribute packets later during network processing could help achieve a more parallelized network stack. Unfortunately, existing software-based packet steering schemes yield minimal performance gain at high communication costs. This paper proposes Interrupt Avoidance Packet Steering (IAPS), a novel packet steering scheme that reduces software-based packet steering overhead. IAPS decreases communication costs by avoiding hardware interrupt triggers during packet steering. IAPS increases application throughput by up to 4.3% and the packet per hardware interrupt ratio by up to 4x.**

***Index Terms*—Packet Steering, Network Processing, RPS/RFS, Hardware Interrupts**

I. Introduction

Packet steering is a discipline in which network packets received by a host machine are distributed onto different cores for further processing. Its purpose is to scale out and distribute network processing in multicore systems. While originally a software-based discipline, nowadays, packet steering has been widely offloaded to Network Interface Cards (NICs).

Hardware-based packet steering has the advantage of distributing packet processing load at no cost to the operating system. Despite this significant advantage, hardware-based packet steering lacks a unique property of software-based packet steering: The ability to *re*distribute packets. The redistribution of packets at the software level could have intriguing application scenarios, such as disaggregating the network processing pipeline into smaller parts, which could enable more fine-grained resource scaling, which has been named one of the big challenges in the current Linux network stack [1].

However, compared to hardware-based solutions, software-based packet steering incurs impermissible communication overhead. Software-based solutions utilize hardware interrupts (hardirqs) to notify target cores of incoming packets to guarantee the highest possible reactivity. Unfortunately, hardirqs lead to unpredictable latency inflation in interrupted processes, so their use is generally frowned upon.

Yet, hardirqs in network processing have been accepted as a necessary evil, and the challenge of reducing them while retaining their benefits has been the target of research in the past [2], [3]. Still, the challenge of reducing hardirqs during software-based packet steering remains open.

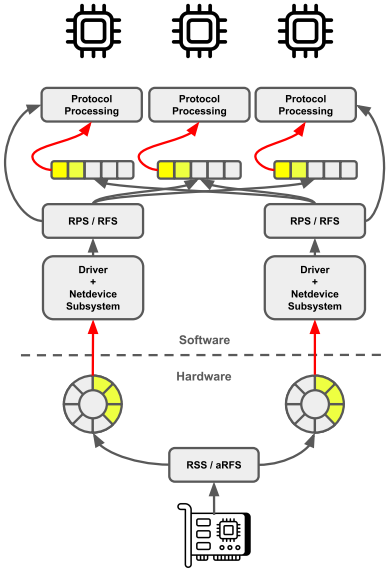


Fig. 1: System with three CPUs and two NIC receive queues using software- and hardware-based packet steering.

This paper introduces Interrupt Avoidance Packet Steering (IAPS), a novel packet steering algorithm that prioritizes minimizing the number of required hardirqs during software-based packet steering. IAPS uses the knowledge of hardirq triggers during packet steering to prioritize selecting cores that will avoid said triggers. We evaluate IAPS alongside other packet steering algorithms in a high-throughput, high-contention environment to see whether a) IAPS successfully reduces the number of hardirqs, and b) decreasing hardirqs results in significant performance increases. Our results show that IAPS increases the number of packets being processed for every hardirq by a factor of four while increasing application throughput by up to 4.3%.

II. Background & Motivation

A. Packet Steering

Packet Steering in the current Linux kernel can occur at two levels: hardware and software. Hardware-based packet steering

is performed directly at the NIC. Upon arrival of a new packet, the NIC enqueues it in one of its receive queues (RX queues). This is where the hardware-based packet steering happens.

If a NIC has only one RX queue, no packet steering is required. If there are multiple RX queues, the decision of which queue to send the packet to is made using a packet steering algorithm. This process is shown in the lower part of Fig. 1. Two packet steering schemes operate at this level: Receive Side Scaling (RSS) and accelerated Receive Flow Steering (aRFS). Once packets have entered an RX queue, the NIC sends a hardirq to the CPU that is associated with the queue. Since an RX queue is only ever associated with one CPU at a time, the packet steering algorithm chooses not only the target RX queue but also the target CPU processing the interrupt. These cores will be referred to as interrupt processing cores. Once the interrupt processing core receives an interrupt from the NIC, it executes an interrupt handling routine which eventually leads to the driver-level and netdevice subsystem-level packet processing. At the end of this stage, software-based packet steering is performed.

The interrupt processing core applies a packet steering algorithm to determine the target CPU for the current packet. Packet steering schemes operating at this level are Receive Packet Steering (RPS) and Receive Flow Steering (RFS). If the target CPU is the interrupt processing core itself, it proceeds to protocol processing. Otherwise, it enqueues the packet on a per-CPU backlog. This process can be seen in the middle of Fig. 1. When a packet is the first to enter a target CPU's backlog, the interrupt processing core schedules the target CPU to receive an Inter-Processor Interrupt (IPI). Once the interrupt processing core is done with its tasks, it sends an IPI to every scheduled target CPU. This triggers a hardirq on the target CPU. Those CPUs will be referred to as protocol processing cores. The protocol processing core will execute an interrupt handling routine, after which it begins further protocol processing. Packets that have been processed are finally enqueued on a socket's receive queue to await reception by the application.

B. Packet Steering Algorithms

Packet steering algorithms define how a target core for packet steering is chosen. There are two packet steering algorithms in common use: One used by RSS/RPS and one used by RFS/aRFS.

RSS/RPS uses a simple packet steering algorithm. Its goal is to evenly distribute the processing load among available CPUs while assigning packets from the same connection to the same CPUs. The algorithm is based on a hash value calculated over the packet's 5-tuple. The hash is used to perform a table lookup, which produces the target CPU. This algorithm offers good load distribution and keeps the target selection simple.

The goal of the packet steering algorithm used in RFS/aRFS is to improve the locality of network and application processing. The algorithm selects the target CPU by checking which CPU performed the most recent receive action from the application. It relies on auxiliary code in the receive system call, which - upon successful packet reception - updates a flow table utilized by the algorithm to identify the application core. By steering packets to the application core, cache locality can be improved, while also guaranteeing that a packet will not be processed on a remote NUMA node.

C. Problem Statement

Existing software-based packet steering schemes are dated. They select target CPUs based on criteria that are not crucial anymore in modern systems.

RPS tries to evenly distribute packets across all available cores, but since it is a software-based packet steering scheme, it only distributes the protocol processing part. This part has been shown to only incur minor overhead compared to other aspects of network processing [4]. Therefore, prioritizing it to be processed by as many cores as possible is unnecessary. Spreading out network processing evenly across all target cores has furthermore been shown to lead to higher tail latencies as opposed to fully saturating a few cores [5].

RFS, on the other hand, steers packets to the application core. At the software level, this would only yield minor benefits. Since the initial reception of the packet has already been performed by the interrupt processing core, all data structures required for network processing will have been allocated there. Therefore, a large part of network processing will have already been performed on a core that might have poor locality to the application core. Furthermore, since the data copy from the kernel to userspace performed by the application core mostly incurs the largest overhead during network processing [4], steering a packet to an application-remote, NUMA-local core has been shown to yield better performance than steering it to the application core [1].

In short, existing packet steering algorithms are better suited for hardware-based packet steering than software-based packet steering, which is why the former replaced the latter. We believe that software-based packet steering can be repurposed to achieve new goals such as network processing disaggregation, but for this, it needs a novel packet steering scheme that addresses the main obstacle: Communication overhead.

III. Proposed Solution

To decrease the overhead incurred by frequent IPIs, we propose IAPS, a novel packet steering scheme that prioritizes steering packets to cores in a way that minimizes the number of necessary IPIs. In this section, we first define the packet steering algorithm used by IAPS. Then, we describe how to manage the data needed for the algorithm's decision-making.

A. IAPS's Packet Steering Algorithm

When selecting a core, we want to select the one that will not result in triggering an IPI as often as possible. Software-based packet steering refrains from triggering an IPI only if it knows that the new protocol processing core has already been notified of the arrival of new packets. This is implemented by enqueueing every new packet on an input queue - one part of a CPU's backlog - and only sending an IPI if the

length of the queue before insertion is 0, i.e. the packet is the first packet to be enqueued. Any following packet that is put on the input queue will be processed as part of the protocol processing triggered by the first packet's IPI. Therefore, by actively steering packets to cores that already have packets enqueued in their input queue, the number of IPIs can be effectively reduced. A core that has at least one packet in its input queue is henceforth referred to as a busy core. When choosing a target core, it is important to not just assign a packet to any of the busy cores. When steering packets to random busy cores, two packets of the same connection could be enqueued on two different cores at the same time. If the processing on those cores proceeds at different speeds, packets of the same connection could be processed out-of-order which can lead to overheads in transport protocols that guarantee in-order delivery such as TCP. To avoid this, IAPS prefers steering packets to the core that processed the prior packet of the same connection, provided that said core is still busy. Suppose the previous core is not busy anymore. In that case, it implies that any prior packet has already started being actively processed, so steering the packet to another busy core will not result in out-of-order processing. Should no busy core be available, the algorithm defaults to RFS's approach of steering the packet to the application core.

B. Busy Core Management

As established in III-A, steering a packet to a core that already has work enqueued reduces IPIs. However, because packets are processed rapidly, checking the length of every CPU's input queue each time the algorithm is executed is too slow. Therefore, busy cores need to be proactively monitored, so that when the algorithm runs, information on busy cores is already available. This is achieved by maintaining a list of busy cores. With a list, the algorithm only has to check whether the list is empty or not to know if a busy core is available. If the list is non-empty, any of the cores can be chosen as the target. This keeps target selection from becoming unreasonably complicated. Cores are inserted into the busy list when the first packet is enqueued on their empty input queue and removed when the input queue has been emptied again. To prevent race conditions on the busy list, all operations are guarded with spin locks. Since all operations on the list are very quick, no significant synchronization overhead can be observed.

IV. Evaluation

A. System Information & Experimental Setup

All experiments were conducted on two machines directly connected over a 100Gbps link. Both machines have an Intel Core i9-10980XE processor with one socket and 18 physical cores, so results only apply to non-NUMA environments. Both are equipped with an Intel E810-C 100G NIC. Though the systems are equipped with 100G NICs, line rate cannot be achieved, because each NIC port is connected using 8*8GT/s PCIe lanes, limiting the effective maximum throughput to a little over 50 Gbps [6]. The Ubuntu version is 20.04.6 and the Linux Kernel version is 6.10.8, with the receiver running a modified version that includes the IAPS implementation.

Experiments were run using iperf version 3.17.1 on both the sender and receiver. During all experiments, the irqbalance service was disabled. To force earlier bottlenecks, both packet steering and application processing have been limited to eight physical cores. RSS is evaluated with eight NIC RX queues, each mapped to one of the eight physical cores. All software-based packet steering schemes are evaluated with only one NIC RX queue. Each experiment was run 10 times and the values in all figures represent the mean over those runs.

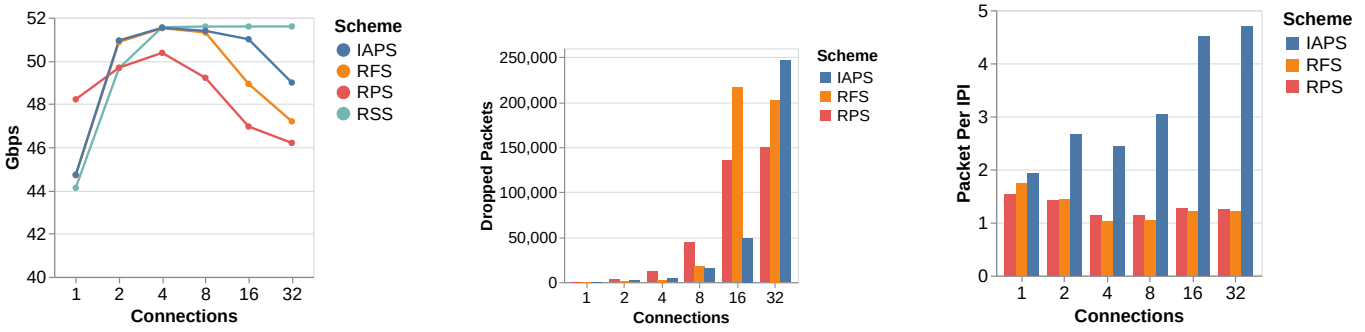
B. Throughput Comparison

To evaluate the effectiveness of IAPS, we compared its performance with other packet steering schemes. We observe the throughput at which the receiver can process incoming traffic as the number of connections increases.

RSS is used as a baseline. It steers packets at the hardware level so traffic is evenly distributed across cores without incurring any communication overhead. As seen in Fig. 2a, RSS reaches a saturation point at four connections and maintains a steady throughput even as connections increase to 32. All of the software-based algorithms also reach their peak at four connections, after which they start declining. Because the software-based schemes only have one CPU acting as the interrupt processing core, it eventually becomes the bottleneck. This is corroborated by the number of dropped packets shown in Fig. 2b. We verified that the drops are caused by an overflowing RX queue. Up until four connections, packet drops are rather sporadic. After that, packet drops increase significantly, leading to a decline in throughput. Notably, IAPS's packet drops spike later than RFS or RPS. This seems to be part of why it maintains a throughput of above 50 Gbps up to 16 connections, as opposed to RPS or RFS, which drop below 50 Gbps at eight and 16 connections, respectively. This suggests that IAPS has a positive effect on the interrupt processing core's per-packet processing time since the only reason the RX queue overflows later than during other packet steering schemes is an interrupt processing core being able to process the packets in the RX queue faster. However, the lower number of packet drops is not the only reason for the IAPS's performance improvement. At 32 connections, Fig. 2b shows that IAPS incurs the highest number of packet drops, yet its throughput is still higher than RPS or RFS with an improvement of 4.3% and 4.2%, respectively. The average number of packets processed for each IPI in Fig. 2c explains this. For any number of connections except for one, IAPS processes more packets per IPI than RPS or RFS. At its maximum at 32 connections, it processes near 4x as many packets per IPI as the other schemes. This shows that IAPS achieves its goal, it also proves that IPIs do affect a system's performance when using software-based packet steering.

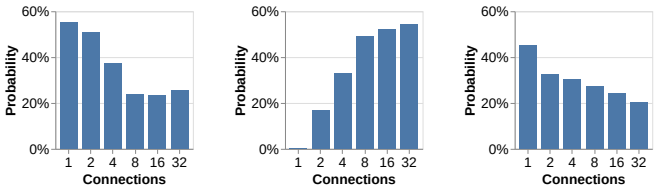
C. Algorithm Effectiveness

To better understand how successful IAPS works, we monitored its target core selection during the experiment. When



(a) Throughput per connection (b) Dropped Packet Count (c) Packet per IPI per connection

Fig. 2: Results for different Algorithms



(a) Previous still busy (b) Choose new busy core (c) No busy core available

Fig. 3: Percentage of steering decisions made by IAPS

IAPS chooses a target core, there are three possible cases:

1) It finds that the core that processed the previous packet is still busy and chooses it as the target (Fig. 3a)
2) The previous target core has gone idle and a new busy target core is found (Fig. 3b)
3) The previous target core has gone idle and there is no other busy core available (Fig. 3c)

Case 1 and 2 avoid IPIs, whereas case 3 would trigger one. The percentages of how often each outcome was chosen by IAPS can be seen in Fig. 3. The general trend is that as server load increases, the likelihood of finding a new busy core increases, while the likelihood of finding no busy core or finding the previous core still busy decreases.

Notably, the likelihood of case 1 and 3 are decreasing at different rates. The percentage of case 1 declines until eight connections and plateaus afterward, while case 3 steadily decreases as connections increase. This suggests that case 1's decline is driven by the increase of maximum busy cores and case 3's decline by the increase in connections. As connections increase, two things happen: First, every connection attributes for a smaller part of the total throughput, and second, the effectiveness of accumulating smaller packets into bigger packets using Generic Receive Offload (GRO) decreases [4]. This means that, assuming a balanced distribution, the load of traffic in Gbps processed by every core will decrease until 8 connections and then stay relatively stable, explaining why case 1's likelihood plateaus. At the same time, the number of total steered packets increases steadily as connections increase, increasing the likelihood of there being at least one busy core available during steering, decreasing the likelihood of case 3.

V. Conclusion

This paper presented IAPS, a novel packet steering scheme that aims to enable the usage of software-based packet steering in new application scenarios by designing a packet steering algorithm to minimize IPI-induced communication overhead. The experiments run to compare IAPS to other packet steering schemes show promise. IAPS was able to increase packet processed per IPI by a factor of four while increasing application throughput by up to 4.3%. We believe that finding new applications for software-based packet steering is an interesting research field and future work will focus on exploring scenarios in which real-life applications can benefit from IAPS.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful feedback. This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2024-00348376, High-Intelligence, High-Versatility, High-Adaptability Reinforcement Learning Methods for Performing Complex Tasks).

References

[1] Q. Cai, M. Vuppalapati, J. Hwang, C. Kozyrakis, and R. Agarwal, "Towards μ s tail latency and terabit ethernet: disaggregating the host network stack," in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022, pp. 767–779.

[2] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond softnet," in *5th Annual Linux Showcase & Conference (ALS 01)*, 2001.

[3] P. Cai and M. Karsten, "Kernel vs. user-level networking: Don't throw out the stack with the interrupts," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 7, no. 3, pp. 1–25, 2023.

[4] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 65–77.

[5] T. Barbette, G. P. Katsikas, G. Q. Maguire Jr, and D. Kostić, "Rss++ load and state-aware receive side scaling," in *Proceedings of the 15th international conference on emerging networking experiments and technologies*, 2019, pp. 318–333.

[6] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding pcie performance for end host networking," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 327–341.