Guided Research Report: Automated Measuring of Ioregionfd and vMux Performance

Sandro-Alessio Gierens
Chair of Distributed Systems & Operating Systems
Department of Computer Science
TUM School of Computation, Information, and Technology
Technical University of Munich

Abstract

Still trailing behind advances in CPU and memory virtualization, I/O virtualization is a common subject to research and development. No matter if it is low-level emulation optimizations like ioregionfd or novel mediation techniques like vMux, both require testing under realistic conditions to hope for adoption in production environments. Virtualized networking is an area where this is particularly difficult due to the large number of influencing factors, high complexity of the setup and huge amount of possible configurations. For evaluation of our previous work on ioregionfd we developed a tool called autotest that automates load latency measurements on specific virtualized networking devices with robustness and efficiency in mind. In this report we will go into more detail about its design and implementation, and present how we extended it for experiments in context of the vMux project. We review measurement results for both ioregionfd and Intel® E810 VFIO devices which are used in the vMux project. Finally we discuss how we plan to scale our experiments to multiple guests to measure virtualized network performance under realistic cloud host conditions.

1 Introduction

Hardware-assisted virtualization extensions in processors have become the de-facto standard for CPU (central processing unit) and memory virtualization [1–7]. Similar techniques exist for I/O (input/output) virtualization, like PCI(peripheral component interconnect)-passthrough and SR-IOV (single root I/O virtualization), they however often trade off flexibility needed in production systems [8–16]. The much less performant emulation remains the default choice for I/O virtualization [17–19] and continues to be a major target of optimization efforts. A current example would be ioregionfd, a file-descriptor-based communication channel between KVM (kernel virtual machine) and a device emulator like QEMU (quick emulator) [20–23], which was also subject of our recent work [24], where we tried to leverage it for improving the MMIO (memory-mapped I/O) performance of Virtio network devices. Other angles to tackle this trade-off are improvements in the interoperability between physical and emulated NICs (network interface cards), like smart NICs that implement the Virtio standard [25–28], but also efforts in the direction of multiplexing between different I/O virtualization techniques, like our current vMux project [29] which aims to provide transparent mediation between typical mediated devices and emulated ones for better scaling and load balancing in dense cloud environments.

The evaluation of such projects requires consistent and generalizable performance measurements which can be challenging, especially in the context of networking. In most non-trivial cases there are at least two or three systems involved, a sender, a receiver, and a network interconnect. With that the number of factors influencing the performance of the whole process increases quickly. This is even more true when virtualization gets involved. Then on top of the plethora of host system choices, there comes another configurable layer of guests. Suddenly there are multiple schedulers at work, virtual CPUs might switch between physical CPUs causing cache misses, and multiple virtual machines might compete for the same resources, other physical machines might compete for the same network resources, and so on. This makes isolating the performance of a single transmission process increasingly difficult. While there are quite a few studies on public cloud performance [30–34], they often lack insights into the underlying mechanisms implemented by the provider. Others offer highly precise measurements and a great deal of detail, but deliver those by simplifying the setup or eliminating sharing altogether [35, 36]. While this is a clean and valid method to evaluate especially the maximum performance of individual components, it also creates a rather artificial setting in contrast real world scenarios like cloud environments where sharing happens on a massive scale. Another issue is the reproducibility of the measurements which is key for establishing the tested techniques. Automation is the obvious solution, but the complexity of virtualization setups requires elaborate scripting for proper experiments.

For our previous work on ioregionfd [24], we therefore developed a tool called autotest (automated testing), which allows us to specify lists of parameters from which it automatically sets up the virtual machine and runs the MoonGen-based L2 (layer 2) load latency measurements accordingly [37, 38]. In this work we will delve deeper into the details of its design and implementation, and report on how we extended it to support our current test system infrastructure. We will also present and discuss the results of more current experiments with ioregionfd as well as during the development of vMux.

This report is structured as follows: In section 2 we introduce the background or mediated virtualization, its advantages over other state-of-the art solutions. We also recapitulate ioregionfd and furthermore establish the foundations of network measurements with special focus on the packet generator MoonGen. In section 3 we first layout the rough design of the system, before diving into the details of the implementation. In section 4 we first discuss the remeasurement of our previous experiments on ioregionfd for Virtio network devices before going over to Intel® E810 VFIO (virtual function I/O) as well as vMux measurements and how we need to adjust our test methodology for simulating realistic cloud host conditions. We summarize our findings in section 5 before giving a brief outlook on future work.

2 Background

This section will introduce the technical background necessary for understanding the remainder of this work. The first 2.1 part will address I/O virtualization, beginning with a comparison of passthrough and emulation, before moving to mediation and the vMux project. In the second part 2.2 will recapitulate the foundation of network performance measurements with state-of-the-art software packet generators focussing on MoonGen and load latency measurements in virtualized environments.

2.1 I/O Virtualization

While hardware-assisted techniques dominate CPU and memory virtualization due to their superior performance and ubiquitous availability in modern processors [1–7], the virtualization of I/O devices remains a divided field [17–19], with roughly three different approaches: passthrough, emulation, and mediation.

2.1.1 Passthrough and Emulation

Passthrough means that a virtual machine gets direct access to a physical device. While this basically leads to native device performance it also limits flexibility, as only one virtual machine can use the device at a time. Devices may also carry a hidden state that cannot be exported, making migration between even the same type of physical device, thus also between hosts, impossible. Security is another concern, since the direct access can pose a threat to the device or even the host system. [39–43]

The capabilities are reversed when we look at the other end of the spectrum. Emulated devices are most entirely implemented in software and therefore usually not bound to the hardware of the host system. This allows for a high degree of flexibility and security, but also comes at the cost of performance. While emulated NICs like QEMU's virtio-net device for example can achieve competitive throughput when packets are large, they quickly crumble under the pressure of high packet rates. [16, 24, 30].

Nevertheless, the performance of emulated devices is usually sufficient for the use cases of average users of IaaS (infrastructure-as-a-service) clouds, where flexibility and security are of much greater importance for the provider to not loose control over their massive systems. Passthrough or similar approaches are usually opt in features that cost extra. This is why even in the commercial sector, emulated I/O devices continue to be a viable option and a subject to optimization efforts like ioregionfd for example. [17–23].

2.1.2 Guest I/O and ioregionfd

There are two common forms in which the CPU accesses an I/O device: DMA (direct memory access) and MMIO (memory mapped I/O). DMA gives the device direct access to a specific region in memory without involving the CPU, so device and CPU may communicate over it asynchronously. MMIO on the other hand maps device memory into the main memory's address space for the CPU to access it in a synchronous manner. Device data planes are almost always implemented using DMA. MMIO is usually located in the control plane, especially for initial DMA setup. [44–53]

While DMA can be elegantly virtualized using memory pages shared between the guest and the device emulator, MMIO requires more attention from the hypervisor. When using QEMU-KVM, a guest MMIO access causes the CPU to exit back to the hypervisor in kernel space, which passes the request on to QEMU by switching to user space. These CPU mode and context switches are expensive and degrade the performance of the guest. [24, 46–48, 54–65]

ioregionfd is a recently proposed KVM optimization that allows user space programs to bind file descriptors to memory regions by calling a specific IOCTL. The descriptors then serve as communication channel over which KVM can forward MMIO requests more directly to the user space program. Even though optimizations like ioregionfd are helpful, the performance gap to passthrough remains significant and shifts attention to alternative approaches like mediation. [20–24]

2.1.3 Mediation and vMux

A mediated device is a virtualized I/O device that provides direct access to hardware resources to multiple virtual machines. This allows for much higher performance than emulation, while the intermediary layer can still maintain a certain degree of security and flexibility. Mediation can be seen as the middle ground between passthrough and emulation, trying to combine both their advantages while mitigating their disadvantages. As such, its actual implementations can vary widely. [66, 67]

A prominent example of mediation that clearly leans more to the passthrough side is SR-IOV (single root I/O virtualization). It allows a physical device to be split into one so-called physical function (PF) and multiple virtual functions (VFs). The physical function has control over the device and can configure its virtual functions, which are then exposed to a guest. Because the virtual functions are mostly implemented in hardware, SR-IOV can achieve highly performant low latency I/O access. Even though SR-IOV lifts the bar in terms of scalability, the number of virtual functions is still limited. While lower end devices, if they have SR-IOV at all, may only support a couple of virtual functions, high end devices nowadays often support more than a hundred. Despite a stronger isolation of the physical function, the virtual functions remain rather unprotected from each other. SR-IOV also doesn't solve the mobility issue of passthrough, as virtual functions are still tied to the physical device. [8–13, 42, 68]

Due to the increasing number of use cases for virtualizing I/O devices, that don't come with built-in SR-IOV support, the Linux kernel reused its VFIO (virtual function I/O) framework to implement a device mediation with virtual function separation at the driver rather than at the hardware level. This helps to address several of the other issues SR-IOV has. The scalability is not fixed by the hardware, but by the capabilities of the device driver, typically allowing for much more virtual functions. In case the driver is open source, its capabilities may also be extended or customized by the community, greatly increasing flexibility. While faulty device drivers can cause similar security risks as SR-IOV, those vulnerabilities can be patched independently of the hardware. Support for live migration is also much better, as the virtual functions are not as tied to the physical device, even though this sometimes requires additional support from the device and driver, especially for zero downtime migration. While the performance is usually slightly lower than with SR-IOV, it is still competitive and much higher than with emulation. In combination with the increased flexibility, this is a step in the right direction from SR-IOV. [66, 67, 69–74]

While new cards often support much more virtual functions than older models, 256 for an Intel® E810 for example, this is still be a bottleneck in dense cloud environments like for FaaS (function-as-s-service), where a single host machine may run more than a thousand guests. While there are other options like VMDq (Virtual Machine Device Queues) which can scale to over 2000 queues, this is a far more rudimentary approach that trades of performance. This is where the idea of our vMux project [29] to multiplex between difference solutions would offer both scalability and balanced performance for guests which require it at the time. [75]

2.2 Network Performance Measurements

When it comes to the performance of a network interface, the first metric that comes to mind is the throughput, which is the amount of data that can be transmitted over the network in a given time, usually measured in Megabits or Gigabits per second (Mbps or Gbps). Throughput is ultimately limited by the so-called line rate, which is the maximum amount of data that can be transmitted over the given link, typically 1, 10 or even 100 Gbps. While just considering the throughput may be fine for comparing network cards in a specific scenario, it leaves out one important aspect that is crucial for a general judgement: the packet size. Any decently advertised network card will be able to achieve the line rate when transmitting large packets, like the typical 1500 byte MTU (maximum transmission unit) or in case of 100 Gbps links only with jumbo frames like 9000 byte. However, within the same link speed class, good network cards will be able to saturate the link even with smaller packets, possibly even minimally sized packets (64 bytes), meaning they are not just good at transmitting packet payloads, but can also quickly process the packet headers. The more suited speed metric is therefore the so-called packet rate, measured in packets per second (pps). [35, 76–78]

Closely tied to the packet rate is the latency, so the time it takes to process a packet. It is usually constant for different packet sizes depending on the NIC and driver implementation, but often increases with higher rates, when packet queues start to get longer. It then bumps up significantly once the line rate or maximum packet rate are reached. This is also when packet loss kicks in. The latency behaviour can therefore be very indicative of a NIC's performance. To not hit the line rate and properly focus on the header processing rather than the payload transfer, one usually uses small packets for latency measurements like 64 byte packets. [35, 76–78]

The typical setup for NIC latency measurements, shown in figure 1, consists of two machines. The load generator (LoadGen) machine uses a packet generator like MoonGen [76] to send load to the device-under-test (DuT) machine, which hosts the network interface card on which the measurements should be performed. The DuT simple reflects the load back to the load generator by swapping the source and destination addresses of the packets. This can be MAC (Media Access Control) or IP (Internet Protocol) addresses, depending on the layer on which the measurement in operated. Mixed in the the load are PTP (Precision Time Protocol) packets that are timestamped by the load generator's NIC on transmis-

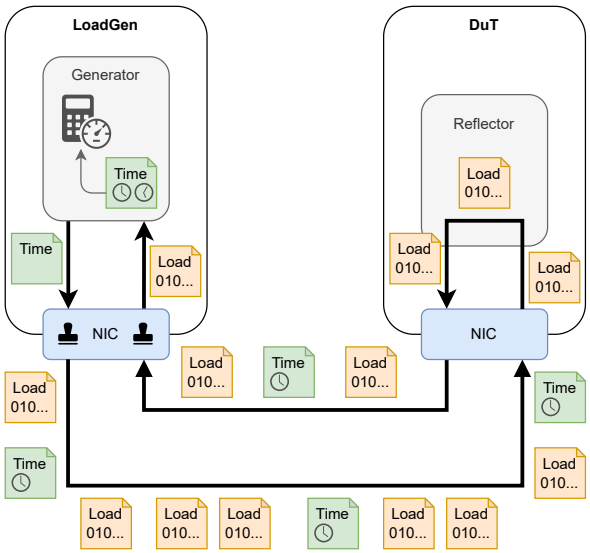


Figure 1: Common setup of NIC latency measurements with a software packet generator like MoonGen. The load generator (LoadGen) machine runs the generator to send out many load and some timestamped packets on the wire. The device-under-test (DuT) machine uses a reflector to bounce every frame back to the LoadGen machine, where the measurement frames get timestamped again. The generator then calculates the round trip time (RTT) from the difference of the departure and arrival timestamps and approximates the latency with this value. [35, 76]

sion and receival. The difference of those timestamps is used by the generator to calculate the round trip time (RTT) to approximate the latency. [35, 76–78]

In case the DuT is a virtual machine, there are several potential complications to consider. Depending on the type of virtual NIC, there might be more layers the traffic passes through, which can not only increase the latency, but also the jitter, which is the variation of the latency over time. Measurement values in general tend to fluctuate more in virtualized environments. Besides the overhead another potential issue is the sheer increase in complexity. There are many ways in which a virtual machines can be linked to the network over a physical NIC. The amount of possible configurations and thus the number of possible measurement scenarios is often high. Also the amount of outside influences, like the host system and other virtual machines, can make it hard to isolate very specific aspects in a measurement. While some studies often try to focus on this by trimming down complexity in their setup and remove as many disturbances as possible for example by pinning vCPUs (virtual CPUs) or I/O processes, this can make the results hard to generalize to setups found in productions systems in the industry. Clouds typically host many different virtual machines on a single physical machine, with different resource requirements and utilizations. Performance can vary significantly depending on the load of the shared resources, so that the customer can not rely on a specific performance level while the provider cannot necessarily guarantee or properly regulate it either. This has been analyzed by many studies, but they can often only get into so much detail, as the lack of insight into or control of the underlying system allows. [35, 76–87]

3 Design and Implementation

In this section we first establish the design goals for our system to make proper performance measurements in the context of ioregionfd and the vMux project in subsection 3.1. We then provide a high-level overview of the system before going into more detail about the implementation in subsection 3.2 to explain how we realize our objectives.

3.1 Design Goals

The main objective of our system is to take comparative performance measurements of specific QEMU network interface setups and their host NICs with focus on load latency. In this context we identify five design goals for our system which are compiled into the following list with no particular order together with a short motivation for each of them:

- **Automation:** Taking measurements for various packet sizes and rates in a statistically significant way is a highly repetitive and tedious task and therefore prone to human error. To ensure consistent and reproducible results, as much as possible should be automated. This includes the setup of the test environment, the execution of the tests and at least partly the analysis of the results.
- **Configuration:** With different interface setups, different QEMU builds and different settings, the automation needs to be accompanied by a flexible and yet fine-grained control over what exactly is being tested. We want to be able to quickly specify a large test matrix, but also not waste time on unneeded combinations.
- **Efficiency:** With the amount of parameters the required time for an entire test run grows quickly. The system should therefore be efficient about setup and test execution and not repeat what has already been done.
- **Robustness:** No system is error free and with the length of test runs the likelihood of unforeseen external disruptions increases. The system should be able to continue where it left off after any sort of intentional and unintentional interruption. This closely ties into the efficiency goal.
- **Scalability:** Because a single virtual machine does not provide a realistic scenario in terms of cloud computing,

we want to be able run and produce network load with multiple machines.

3.2 System Layout and Implementation

When it comes to automation on Linux systems there is a multitude of options from BASH scripts to program command line workflows to frameworks like Ansible for orchestrating entire clusters. We need to manage multiple machines which can be cumbersome in BASH, but also not on the scale that makes Ansible necessary. On top of that we require a certain level of abstraction and data processing, therefore we decide to use Python as it offers us a lot of flexibility in those areas.

Our program, with the rather unimaginative working title **autotest** for "automated testing", is structured into three modules: `autotest`, `server` and `loadlatency`. The main module `autotest` contains the `main` function as entry point to the program. It uses `argparse` to define several commands and handle their arguments. The following list provides a short initial overview of those commands. They are explaining in more detail below.

- `ping`: Ping all servers, loadgen, host and guest to see check they are all reachable.
- `run-guest`: Start the guest on the host machine.
- `kill-guest`: Stop the guest running on the host.
- `setup-network`: Just setup the network for the guest on the host without running the guest.
- `teardown-network`: Teardown the guest network on the host.
- `test-load-lat-file`: Run load latency tests defined in a test config file.
- `acc-load-lat-file`: Force accumulation of all load latency tests defined in a test config file.
- `shell`: Enter a Python3 shell with access to the server objects for debugging and manual intervention.
- `upload-moonprogs`: Upload the MoonGen programs to the servers.

After parsing the command line arguments, `autotest` sets up logging using the `logging` and `colorlog` modules based on the requested verbosity level (error by default, and up to debug with the `-vvv` argument). After that it reads the main configuration file `autotest.conf` using the `configparser` module. This file contains sections for each of the servers, host, guest and loadgen, with NIC information like PCI-bus and MAC-addresses, driver information and paths like for the QEMU build, the packet generator and so on. Those information is used to instantiate corresponding `Host`, `Guest` and `Loadgen` objects defined in the `server` module. They are all derived from the `Server` class to provide a generic interface for common tasks like executing commands, up- and downloading files and managing TMUX sessions for running QEMU or MoonGen in the background. The subclasses extend this by type-specific functionality like running the guest on the host or starting and stopping the load generator on the loadgen server. The command execution on which most of the other functionality is based is implemented such, that it automatically detects whether the server the localhost, and then either uses `subprocess` for local execution or `paramiko` for execution via SSH on the remote server. This means `autotest` can be used from the host or loadgen machine, but also some other workstation or laptop improving flexibility. The object-oriented approach for the servers makes extensibility to more complex setups and multiple guests possible.

The actual execution of load latency tests is implemented in the `loadlatency` module. The command `test-load-lat-file` reads the configuration from the `tests.conf` file and instantiates a `LoadLatencyTestGenerator` object (which is not a Python generator). These objects takes all the loop parameters and generates a tree where each layer represents one parameter type and each leaf a specific combination of all those parameters. Only valid combinations will be generated though, so a VFIO-passthrough interface will not be combined with a PCI-less MicroVM guest. This full tree is then reduced by removing all combinations for which output files are already present in the output directory allowing re-execution after failure without having to rerun anything. The order or the loop parameters in the tree is such, that parameters that are deeper ingrained in the setup are closer to the root of the tree. This means the generator loops over parameters like the packet size or rate close to the leaves while operating on the same virtual machine, and only changing it when it actually comes to iterations of parameters like network interface or machine type closer to the root that actually require setting up a new virtual machine for example. This minimizes setup overhead therefore adding further to the efficiency of the program.

Building the full tree from all possible combinations of the given parameters makes it possible to setup large test cases by just writing out a few lists of parameters in the configuration file. To leave out a specific value it just needs to be removed from the parameter list. The test configuration file can contain multiple sections with different test cases which are then just generated and executed one after the other. By keeping the same output directory in multiple sections, it is possible to create more complex cases where only specific combinations of parameters are allowed. Say we want to to measure close to the load maximum of a physical NIC and emulated NICs on top of it. Since the emulated interfaces will have a much lower performance measuring both interface types with the same range of packet rates would be a waste of time. The physical NIC would not saturate with the highest rates the

emulated NICs can handle, while the emulated NIC overloads with the rates of the physical NIC. Therefore it is prudent to define two test sections, one for the physical NIC and one for the emulated NIC with appropriate ranges of packet rates.

For the load generation and reflection we use MoonGen, a DPDK-based packet generator (Data Plane Development Kit [88]) that uses a NICs hardware timestamping capabilities to acquire high precision latency measurements. It can saturate 10 Gbit/s links with minimally sized packets and runs test programs written in Lua allowing to deeply customize test scenarios. In particular we use a custom version of MoonGen's sample program `l2-load-latency.lua` [89] and also a custom version of libmoon's `reflector.lua` [**?**] on the other side. The test configuration also contains a variable for setting the number of repetitions. If it is greater than one, the resulting latency histograms will be accumulated while values like the throughput can be averaged. This way the statistical significance of the measurements can be increased and the effect of disturbances reduced. It is also possible to do warmup and cooldown runs before the actual measurement. While the details of the `loadlatency` modules are specific to MoonGen load latency measurements, the overall structure of the the test generator is useful in many cases. By adding server functions for those and generalizing this module, the system can be extended to other types of measurements.

4 Evaluation

After diving into the rough implementation in the previous section, we now come to the actual application of the software. We first report on the migration to the chair servers and the repetition of our previous work's measurements with focus on ioregionfd. Then we come to the short implementation and evaluation of ioregionfd's posted writes mode. After that we discuss measurements on the Intel® E810 NIC, in particular VFIO passthrough in more detail as well as bottlenecks that we face and how one may overcome them. Finally we lead over to the preparation regarding vMux and draft dense multi-VM experiments in order to evaluate its performance under more realistic conditions.

4.1 Virtio Remeasurement

In our previous work [24] we implemented ioregionfd in QEMU's Virtio framework and evaluated it using autotest. While we were seeing very small changes in load latency behaviour, this was mostly overshadowed by significant packet losses of at the time unclear origin. Despite a fuzzy trend towards higher losses with ioregionfd, we deemed the results too inconclusive to make a statement about the performance characteristics of ioregionfd. All measurement data and plots can be found here [90, 91].

At the time our test system consisted of two older Intel® Xeon® E5 based servers with Intel® 82599ES 10 Gb/s NICs interconnected via a 10 Gb/s coaxial direct attach cable (DAC), see [24] for details. To rule out hardware problems and since 100Gb/s NICs are used in context of the vMux project, we first migrate to a new test system. It consists of two Intel® Xeon® Gold 5317 CPU @ 3.00GHz dual socket powered SuperMicro® servers (named river and wilfred) with each one Intel® E810-C for QSFP (quad small formfactor pluggable) 100 Gb/s NICs connected with coaxial DACs over a switch. For more details on the hardware of the hosts refer to [92, 93]. The biggest difference likely is moving from Debian 11 bullseye to NixOS 22 Raccoon which entails a couple of changes to our testing program.

With this setup we repeat all the Virtio measurements from our previous work. All output data and plots can be found here [94, 95]. All relevant plots are also included in the appendix 6. Here we will only discuss parts of that and mostly focus on noticeable differences to our previous measurements [24].

4.1.1 Reflector

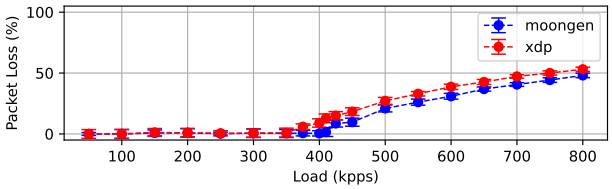


Figure 2: Packet loss comparison between MoonGen and XDP reflector with MacVTap-based Virtio PCI network device with vhost protocol off and packet size 64 Bytes. The whole plot set can be found in figure 9.

Figure 2 shows an example of a packet loss comparison between a MoonGen- and XDP(express data path)-based reflector. The general trend is that even when we see higher latencies with the MoonGen reflector, it is still reaches a higher load limit than the XDP reflector. This is similar to our previous results [24].

4.1.2 Interface

The comparison plots between bridge and MacVTap interfaces are shown in figure 10. There are no notable differences to our previous measurements [24]. Even with sometimes higher final latency the MacVTap reaches a slightly higher load limit.

4.1.3 Bus Type

Figure 3 shows an example comparison of load latency and packet loss between a PCI-based guest and a MicroVM. In contrast to our previous measurements [24] the PCI-based

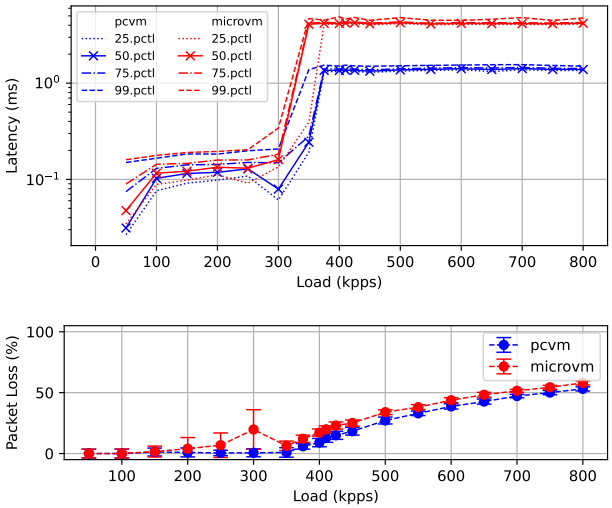


Figure 3: Load latency and packet loss comparison between PCI-based guest and MicroVM with MacVTap-based Virtio network device with vhost protocol off and packet size 64 Bytes. The whole plot set can be found in figure 11.

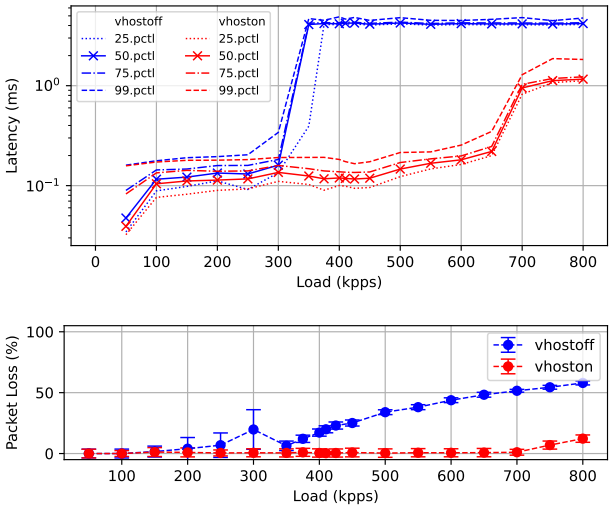


Figure 4: Load latency and packet loss comparison between vhost on and off with MacVTap-based Virtio network device in a MicroVM guest and packet size 64 Bytes. Full plot set can be found in figure 12.

guest outperforms the MicroVM in all cases in terms of load latency. For the packet loss it is ahead when vhost is off but slightly behind when vhost is on. This is difference is only marginal and while the PCI guest has zero packet loss before the load limit, the MicroVM has. This is however be no means as pronounced as in our previous measurements [24].

4.1.4 Vhost Protocol

Figure 4 shows an example comparison of load latency and packet loss between vhost protocol turned on and off. While we where unable to make out any difference in our previous work [24] we can now see a clear performance benefit from vhost. The load limit is almost doubled and the final latency is significantly lower. Also while without vhost we see some packet loss near the load limit, vhost eliminates it completely.

4.1.5 Ioregionfd

Figure 5 shows the comparison example between ioregionfd turned off and applied to the Virtio network device's interrupt status register as well as all registers that work with ioregionfd in a MicroVM guest with MacVTap interface and vhost protocol on. Also in this case vhost seems to be able prevent any packet loss. Our assumption is that the newer CPUs of our current test setup are in contrast to the previous one fast enough to take advantage of the vhost protocol. As other measurements show, see figure 13, ioregionfd again appears to increase packet loss in some cases. As for the latency however it does not produce a measurable difference in any case. This means we can now conclusively rule out that a potential performance benefit of ioregionfd was hidden by the unexplained packet loss in our previous work [24].

4.2 Ioregionfd Posted Writes Mode

One ioregionfd feature we have not looked at in our previous work is its posted writes mode. This mode makes all write accesses handled by the particular descriptor to be posted, meaning the access immediately returns to the guest without waiting for the actual write to complete. We therefore add the ability to enable this mode on a per-descriptor basis to our existing ioregionfd QEMU patch [96]. Figure 6 shows an example comparison of load latency and packet loss between ioregionfd turned off as well as applied with and without posted writes mode to either all registers or just the interrupt status register in a MicroVM guest with MacVTap interface and vhost protocol on.

The interrupt status register is the only register that is accessed regularly during operation or on a per packet basis. However, since it is a read only register, the matching latency with all other configurations is not surprising. More as a sanity check we also apply it to all registers as we have done before. This interestingly still does not change the latency behaviour, but leads to almost 100% packet loss. The reason for this is, that some registers are expected to not return until the write has been completed by the driver, meaning they are incompatible with posted writes. These registers not working prevents a proper device setup causing the packet loss.

While we could try out the posted writes mode and measure it on each register individually, seeing that measuring for the shown configurations alone takes several days, does not seem

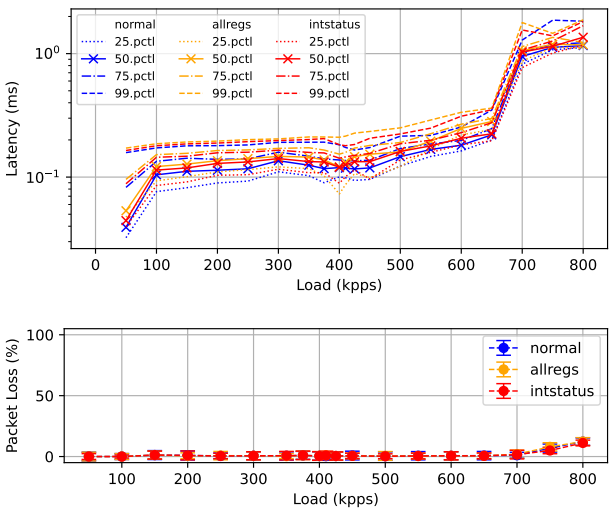


Figure 5: Load latency and packet loss comparison between ioregionfd on (intstatus: applied to interrupt status register, allregs: applied to all registers for which it works) and off with MacVTap-based Virtio network device in a MicroVM guest with vhost protocol on and packet size 64 Bytes. Full plot set can be found in figure 13. For packet size 1024 Bytes see figure 14.

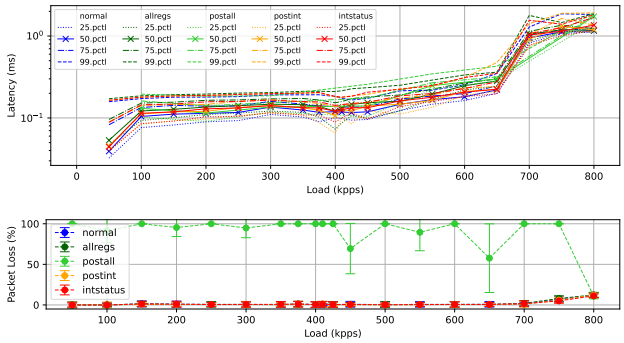


Figure 6: Load latency and packet loss comparison between ioregionfd on (intstatus: applied to interrupt status register, allregs: applied to all registers for which it works) with and without posted writes (postint and postall) and off with MacVTap-based Virtio network device in a MicroVM guest with vhost protocol on and packet size 64 Bytes. Full plot set can be found in figure 15.

worth the effort. We suspect it would not change the latency behaviour and thus simply confirm our tracing results [24] that no other register is accessed regularly.

We therefore conclude that ioregionfd applied to Virtio does not provide any performance benefit but rather potential deficits in terms of packet loss with some configurations. In the remainder of the section we will now leave the realm of emulation and focus on passthrough and mediation devices.

4.3 E810 VFIO and Bottlenecks

Figure 7 shows the comparison of load latency and packet loss between different packet sizes for E810 VFIO passthrough devices into a QEMU guest.

Up to 11 Mpps the latency for all sizes is almost identical at around 10 ns with zero packet loss everywhere. At 12 Mpps the latency for the largest packet size of 1020 Bytes (1024 Bytes with the CRC checksum) jumps up beyond 100 ns and stays there. The reason for this ist simply that for this packet size the line rate of 100 Gbps is reached at roughly 12 Mpps. The packet loss is still measured at zero after this, because we calculate it from the difference in sent and received traffic. Since we reach the line rate, the surplus of packets is never actually send out and thus not accounted for.

Even though the latencies are roughly the same until a jump, it should be noted that they split up marginally over time in accordance with their size, so that a 18 Mpps the 508 Byte latency is about 2.5 ns higher than for 60 Byte and 124 Byte, with 252 Byte roughly in the middle. At 19 Mpps all the remaining latencies then jump up to around 20 ns. Since the line rate is not reached and the packet loss calculation still works, the packet loss also behaves as expected and slowly increases.

19 Mpps is quite low in comparison to the roughly 150 Mpps that would saturate a 100 Gbps link for minimally sized packets. Intel's DPDK performance report [97] moreover shows that a physical E810 card should be able to reach over 100 Mpps for 64 Byte packets. Interestingly even increasing the number of reflection queues to four as in the report does not change the 19 Mpps limit. Suspecting that we do not actually reach a limit of the NIC, we take a closer look at our log files and find that the load generator actually sends out beyond 19 Mpps and the reflector also receives this. To solidify this finding, we let two instances of our customized load generator script [89, 98] send traffic towards the other. We find that we can send and receive about 56 Mpps with a single queue in both directions, and 117 Mpps in total with three or more queues in both directions. This suggests that the reflector [99] is the actual bottleneck. We are in contact with the MoonGen developers about this, but we believe that for the multi-VM scenario we are striving for the current performance might be sufficient.

4.4 vMux and Multi-VM Scenario

At the time of writing this report, there is no fully-functional vMux prototype yet. The current implementation has a few issues with the initial DMA setup. While this makes the virtual NIC unusable, it does not prevent the guest from booting. Therefore we already prepare autotest to setup a guest with this type of device and start the vMux process in a separate

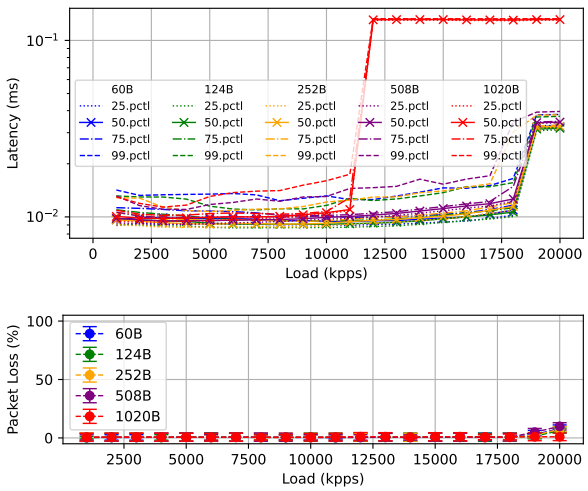


Figure 7: Load latency and packet loss comparison between difference packet sizes for E810 VFIO passthrough devices into a QEMU guest.

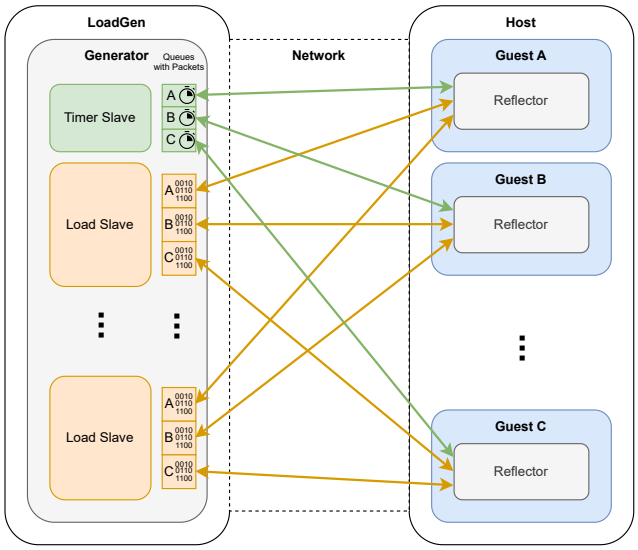


Figure 8: Diagram sketching scale up of out load latency measurements. Host runs multiple guests each with a reflector. The load generator runs one timer slave and potentially multiple load slaves. Each attach to one queue and send packets to every guest.

TMUX session.

Since vMux is not so much about improving the performance of a single guest NIC, but more about better balancing the performance according to multiple guest's needs, the approach we took to testing so far needs ot be adjusted. Like on a real world cloud host machine we want to observe how latency, throughput and packet loss are affected by multiple guests striving for the same resources.

In subsection 3.2 we already discussed that autotest is without too much trouble capable of running multiple guests due to its object-oriented design, making scalability on the host side mostly a coding effort. On the side of the load generator we however face a few more foundational hurdles. In the previous subsection 4.3 we have seen, that the NIC already has to deal with a significant amount of load even with just one VFIO device. We therefore need to assume that multiple load queues are necessary. We could use one queue per target but this does not scale well. Another issue is, that the NIC only has a single timestamp register, so only one queue can use it. Our test design therefore is to have one timer slave process and one or more load slaves, each sending their packets to all guests. Figure 8 sketches this setup.

It should be noted that this setup does not simulate intra-host traffic, so traffic between guests on the same host. We would however argue that inter-host traffic is more common in typical on-demand cloud deployments. Besides the source and destination of the traffic, another important aspect to consider is the packet size. Sofar we have only tested with a single packet size at a time, while real world traffic is usually a mix of sizes. So our system should support inputting a size distribution, maybe even on a per guest basis to make it less homogeneous and more realistic. Unfortunately most studies on this topic are a bit dated [100–108]. They also mostly focus on internet traffic, which does not necessarily reflect traffic on cloud hosts. We therefore might have to do some investigation of our own.

5 Conclusion

This report first discusses how mediation as middle ground of I/O virtualization tries to combine the performance of passthrough and flexibility of emulation with special focus on VFIO as state of the art and a short introduction of the vMux project and ioregionfd. It established latency as key metric for NIC performance and describes how it is commonly measured with packet generators like MoonGen. We then explain how autotest (those codebase can be found here: [38]) automates this process in an efficient manner. We focus on how the test configuration is read into a tree structure which is ordered such, that the tests can be setup and run with minimal overhead, and reduced so that only the still necessary measurements are taken. We furthermore argue how the object-oriented server definitions allow for easy scalability to simulate more realistic conditions.

During the evaluation for our Virtio ioregionfd remeasurement we can further underline some of the results of our previous work, like higher performance of a MoonGen L2 reflector in comparison to an XDP-based solution, but also find notable differences. PCI outperforms the MicroVM machine type in all measured cases and MicroVM also has some but much lower packet loss than on our old test system. We now

also see a performance boost from the Vhost protocol which does not only double the load limit but also eliminates any packet loss. We attribute this mainly to the more recent much faster CPUs in our new test system. Despite the reduction in packet loss we still do not see any measurable difference in latency behaviour from ioregionfd also with the just now measured posted-writes mode. By eliminating most of the uncertainties from our previous work we can now conclude with much more confidence that the application of ioregionfd to Virtio's MMIO registers does not benefit performance but might even be detrimental to it by increasing packet loss.

With Intel® E810 VFIO devices we can easily saturate the 100G link with large 1024 B packets, but unfortunately run into a MoonGen reflector bottleneck with smaller packets at 19 Mpps. By sending and receiving load in both directions and essentially taking the reflector out of the picture, we can confirm that NIC is capable of more than 100 Mpps with minimal sized packets and even reach up to 117 Mpps. In the absence of a fully functional vMux prototype we cannot yet compare its performance in the single VM scenario. We however explain in detail how we plan to use multiple load generator slaves and multiplexing to many targets within each of them to simulate a more realistic cloud host environment with multiple guests and heterogenous packet sizes. Implementing those plans will be the focus of our future work.

References

[1] (2022) About Nested Virtualization. Google LLC. [Online]. Available: https://cloud.google.com/compute/docs/instances/nested-virtualization/overview (Accessed 2022-08-21).

[2] (2021) Getting started with KVM. IBM Corporation. [Online]. Available: https://www.ibm.com/docs/en/cic/1.1.3?topic=SSLL2F_1.1.3/com.ibm.cloudin.doc/overview/Getting_started_tutorial.html (Accessed 2022-08-21).

[3] (2022) Oracle Virtualization. Oracle. [Online]. Available: https://www.oracle.com/virtualization/ (Accessed 2022-08-21).

[4] S. Sharwood. (2017) Aws adopts home-brewed kvm as new hypervisor. [Online]. Available: https://www.theregister.com/2017/11/07/aws_writes_new_kvm_based_hypervisor_to_make_its_cloud_go_faster/ (Accessed 2022-08-21).

[5] Openstack. (2021) Most commonly used OpenStack compute (Nova) hypervisors worldwide, as of 2020. [Online]. Available: https://www.statista.com/statistics/1109443/worldwide-openstack-hypervisors/ (Accessed 2022-08-21).

[6] C. Aker. (2015) Linode turns 12! Here's some KVM! Linode LLC. [Online]. Available: https://www.linode.com/blog/linode/linode-turns-12-heres-some-kvm/ (Accessed 2022-09-10).

[7] C. Aker. (2016) KVM Update. Linode LLC. [Online]. Available: https://www.linode.com/blog/linux/kvm-update/ (Accessed 2022-09-10).

[8] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," *J. Parallel Distrib. Comput.*, vol. 72, no. 11, p. 1471–1480, nov 2012. [Online]. Available: https://doi.org/10.1016/j.jpdc.2012.01.020

[9] F. W. Maximilian Fischer, "Survey on sr-iov performance," 2022. [Online]. Available: https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2022-01-1/NET-2022-01-1_09.pdf (Accessed 2021-09-10).

[10] J. Liu, "Evaluating standard-based self-virtualizing devices: A performance study on 10 gbe nics with sr-iov support," pp. 1–12, 2010.

[11] C. Robison, *Configure SR-IOV Network Virtual Functions in Linux* KVM**, Intel Corporation, 2017. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/configure-sr-iov-network-virtual-functions-in-linux-kvm.html (Accessed 2021-09-10).

[12] D. D. Yu Zhao, *PCI Express I/O Virtualization Howto*, 2009. [Online]. Available: https://docs.kernel.org/PCI/pci-iov-howto.html (Accessed 2021-09-10).

[13] W. Han. (2015) Live migration with sr-iov pass-through. Huawei Technologies Co., Ltd. [Online]. Available: http://www.linux-kvm.org/images/9/9a/03x07-Juniper-Weidong_Han-LiveMigrationWithSR-IOVPass-through.pdf (Accessed 2021-09-10).

[14] A. S. Chavan, A. S. Varal, V. S. Bhende, and M. Thalor, "Experimental analysis of dedicated gpu in virtual framework using vgpu," in *2021 International Conference on Emerging Smart Computing and Informatics (ESCI)*, 2021, pp. 483–489.

[15] A. Garg, P. Kulkarni, U. Kurkure, H. Sivaraman, and L. Vu, "Empirical analysis of hardware-assisted gpu virtualization," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019, pp. 395–405.

[16] R. Leira, G. Julián-Moreno, I. González, F. J. Gómez-Arribas, and J. E. López de Vergara, "Performance assessment of 40 gbit/s off-the-shelf network cards for virtual network probes in 5g networks," *Computer Networks*, vol. 152, pp. 133–143, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S138912861930101X

[17] *Enhanced Networking on Linux*, Amazon Web Services, 2022. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html (Accessed 2021-09-10).

[18] *Set up SR-IOV networking*, Google Cloud, 2022. [Online]. Available: https://cloud.google.com/anthos/clusters/docs/bare-metal/latest/how-to/sriov (Accessed 2021-09-10).

[19] *SR-IOV backet networks*, IBM Corporation, 2022. [Online]. Available: https://www.ibm.com/docs/en/powervc-cloud/1.4.4?topic=networks-sr-iov-backed (Accessed 2021-09-10).

[20] S. Hajnoczi. (2020) Proposal for MMIO/PIO dispatch file descriptors (ioregionfd). [Online]. Available: https://www.spinics.net/lists/kvm/msg208139.html (Accessed 2022-03-19).

[21] sbrksb. (2020) A brief introduction to the ioregionfd project. [Online]. Available: https://sbrksb.github.io/2020/12/10/intro.html (Accessed 2022-03-19).

[22] E. Afanasova. (2021) Introduce mmio/pio dispatch file descriptors (ioregionfd). [Online]. Available: https://lore.kernel.org/all/cover.1613828726.git.eafanasova@gmail.com/ (Accessed 2022-08-21).

[23] J. Thalheim and sbrksb. (2021) Comparing changes (linux peter/5.12.14-v0 with v5.12.14). [Online]. Available: https://github.com/vmuxIO/linux/compare/v5.12.14...vmuxIO:peter/5.12.14-v0 (Accessed 2022-03-19).

[24] S.-A. Gierens, "Rethinking I/O Emulation Architectures for VMs," Bachelor's Thesis, Chair for Distributed Systems Engineering, Department of Informatics, Technical University of Munich, 2022. [Online]. Available: https://github.com/TUM-DSE/research-work-archive/blob/main/archive/2022/summer/docs/bsc_gierens_rethinking_io_emulation_architectures_for_vms.pdf (Accessed 2023-04-07).

[25] J. Wang and A. Adam. (2022) Hardening virtio for emerging security use cases. [Online]. Available: https://www.redhat.com/en/blog/hardening-virtio-emerging-security-usecases (Accessed 2023-04-09).

[26] *VirtIO-net Emulated Devices*, NVIDIA Corporation & affiliates, 2021. [Online]. Available: https://docs.nvidia.com/networking/m/view-rendered-page.action?abstractPageId=34265607 (Accessed 2023-04-09).

[27] S. Zagorianakos and J. Tonsing, "Implementing a fourth generation smartnic," in *SmartNICs Summit*, 2022. [Online]. Available: https://smartnicssummit.com/proceeding_files/a0q5f000000lcIn/20220427_A-101_Zagorianakos.PDF (Accessed 2023-04-09).

[28] S. Hajnoczi and M. Tsirkin, "Virtio without the 'virt'," 2019. [Online]. Available: https://lwn.net/Articles/805235/ (Accessed 2023-04-09).

[29] P. Okelmann, S.-A. Gierens, J. Thalheim, P. Sabanic, and K. Mio. (2023) vmuxio/vmuxio. [Online]. Available: https://github.com/vmuxIO/vmuxIO (Accessed 2023-04-09).

[30] D. Ghoshal, R. S. Canon, and L. Ramakrishnan, "I/o performance of virtualized cloud environments," in *Proceedings of the Second International Workshop on Data Intensive Computing in the Clouds*, ser. DataCloud-SC '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 71–80. [Online]. Available: https://doi.org/10.1145/2087522.2087535

[31] B. Hou, F. Chen, Z. Ou, R. Wang, and M. Mesnier, "Understanding i/o performance behaviors of cloud storage from a client's perspective," *ACM Trans. Storage*, vol. 13, no. 2, may 2017. [Online]. Available: https://doi.org/10.1145/3078838

[32] I. Sadooghi, J. H. Martin, T. Li, K. Brandstatter, K. Maheshwari, T. P. P. de Lacerda Ruivo, G. Garzoglio, S. Timm, Y. Zhao, and I. Raicu, "Understanding the performance and potential of cloud computing for scientific applications," *IEEE Transactions on Cloud Computing*, vol. 5, no. 2, pp. 358–371, 2017.

[33] B. Zhang, X. Wang, R. Lai, L. Yang, Y. Luo, X. Li, and Z. Wang, "A survey on i/o virtualization and optimization," in *2010 Fifth Annual ChinaGrid Conference*, 2010, pp. 117–123.

[34] P. Ivanovic and H. Richter, "Openstack cloud tuning for high performance computing," in *2018 IEEE 3rd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, 2018, pp. 142–146.

[35] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart, and G. Carle, "Throughput and latency of

virtual switching with open vswitch: A quantitative analysis," *Journal of Network and Systems Management*, vol. 26, 04 2018. [Online]. Available: https://www.net.in.tum.de/fileadmin/bibtex/publications/papers/Jnasm2017.pdf (Accessed 2022-08-21).

[36] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Performance characteristics of virtual switching," in *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*. IEEE, 2014, pp. 120–125.

[37] S.-A. Gierens. (2022) autotest. [Online]. Available: https://github.com/vmuxIO/autotest/ (Accessed 2022-08-30).

[38] S.-A. Gierens. (2023) vmuxio/vmuxio/test. [Online]. Available: https://github.com/vmuxIO/vmuxIO/tree/main/test (Accessed 2023-04-09).

[39] (2023) Pci passthrough. Proxmox Server Solutions GmbH. [Online]. Available: https://pve.proxmox.com/wiki/PCI_Passthrough (Accessed 2023-04-09).

[40] (2023) Pci passthrough. Red Hat, Inc. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/virtualization/chap-virtualization-pci_passthrough (Accessed 2023-04-09).

[41] I. Red Hat. (2023) Hardening infrastructure and virtualization. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/14/html/security_and_hardening_guide/hardening_infrastructure_and_virtualization (Accessed 2023-04-09).

[42] A. Richter, C. Herber, T. Wild, and A. Herkersdorf, "Denial-of-service attacks on pci passthrough devices: Demonstrating the impact on network- and storage-i/o performance," *Journal of Systems Architecture*, vol. 61, no. 10, pp. 592–599, 2015, special section on Architecture of Computing Systems edited by Editors: Wolfgang Karl, Erik Maehle, Kay Römer, Eduardo Tovar, Martin Danek Special section on Testing, Prototyping, and Debugging of Multi-Core Architectures edited by Editors: Frank Hannig & Andreas Herkersdorf Special section on Embedded Vision Architectures and Applications edited by Editors: Christophe Bobda, Walter Stechele, Ali Ahmadinia and Miaoqing Huang. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762115000739

[43] S. Lee and S. Yoo, "Breaching gpu data on a cloud vm," in *2022 13th International Conference on Information and Communication Technology Convergence (ICTC)*, 2022, pp. 205–207.

[44] R. Williams, *Computer Systems Architecture: A Networking Approach*, ser. Computer Systems Architecture: A Networking Approach. Addison-Wesley, 2001, no. Bd. 1. [Online]. Available: https://books.google.de/books?id=P4cpAQAAMAAJ

[45] D. Serpanos and T. Wolf, *Architecture of Network Systems*, ser. ISSN. Elsevier Science, 2011. [Online]. Available: https://books.google.de/books?id=o7GhrZXM_ssC

[46] A. S. Tanenbaum and T. Austin, *Structured Computer Organization*, 6th ed. USA: Prentice Hall Press, 2012.

[47] M. Rafiquzzaman, *Fundamentals of Digital Logic and Microcontrollers*, 6th ed. Wiley Publishing, 2014.

[48] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[49] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[50] Y. Klimiankou, "Design and implementation of port-mapped io management subsystem and kernel interface for true microkernels on ia-32 processors," *Programming and Computer Software*, vol. 45, pp. 319–323, 11 2019.

[51] A. K. Jack Belzer, Albert G. Holzman, *Encyclopedia of Computer Science and Technology: Volume 10 - Linear and Matrix Algebra to Microorganisms: Computer-Assisted Identification*. CRC Press, 1978.

[52] E. D. Reilly, *Memory-Mapped I/O*. GBR: John Wiley and Sons Ltd., 2003, p. 1152.

[53] A. Papagiannis, M. Marazakis, and A. Bilas, "Memory-mapped i/o on steroids," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 277–293. [Online]. Available: https://doi.org/10.1145/3447786.3456242

[54] E. P. Martín. (2020) Virtio devices and drivers overview: The headjack and the phone. [Online]. Available: https://www.redhat.com/en/blog/virtio-devices-and-drivers-overview-headjack-and-phone (Accessed 2022-04-02).

[55] E. P. Martín. (2020) Virtqueues and virtio ring: How the data travels. [Online]. Available: https://www.redhat.com/en/blog/virtqueues-and-virtio-ring-how-data-travels (Accessed 2022-09-02).

[56] A. K. et. al. (2018) Virtio uses dma api for all devices. [Online]. Available: https://patchwork.ozlabs.org/project/linuxppc-dev/cover/20180720035941.6844-1-khandual@linux.vnet.ibm.com/ (Accessed 2022-09-02).

[57] Peterx. (2022) Features/vt-d. [Online]. Available: https://wiki.qemu.org/Features/VT-d (Accessed 2022-09-02).

[58] I. Corporation, *Intel Virtualization Technology for Directed I/O*, 2022. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf (Accessed 2022-09-02).

[59] olive.zhao. (2021) Comparison of KVM and XEN Technologies. [Online]. Available: https://forum.huawei.com/enterprise/en/comparison-of-kvm-and-xen-technologies/thread/773247-893 (Accessed 2022-08-21).

[60] Terenceli. (2018) Kvm mmio implementation. [Online]. Available: https://terenceli.github.io/%E6%8A%80%E6%9C%AF/2018/09/03/kvm-mmio (Accessed 2022-09-02).

[61] walkero. (2022) Mmio simulation principle. [Online]. Available: https://programmer.ink/think/mmio-simulation-principle.html (Accessed 2022-09-02).

[62] L. T. et. al. (2022) Linux kvm handle_ept_misconfig. [Online]. Available: https://github.com/torvalds/linux/blob/5995497296ade7716c8e70899e02235f2b6d9f5d/arch/x86/kvm/vmx/vmx.c#L5675-L5694 (Accessed 2022-09-02).

[63] L. T. et. al. (2022) Linux kvm vmx_handle_exit. [Online]. Available: https://github.com/torvalds/linux/blob/5995497296ade7716c8e70899e02235f2b6d9f5d/arch/x86/kvm/vmx/vmx.c#L6283-L6483 (Accessed 2022-09-02).

[64] QEMU Project Developers. (2022) Qemu kvm_cpu_exec. [Online]. Available: https://github.com/qemu/qemu/blob/61fd710b8da8aedcea9b4f197283dc38638e4b60/accel/kvm/kvm-all.c#L2869-L3044 (Accessed 2022-09-02).

[65] *The memory API*, QEMU project developers, 2022. [Online]. Available: https://www.qemu.org/docs/master/devel/memory.html (Accessed 2022-03-19).

[66] (2023) Vfio - virtual function i/o. The kernel development community. [Online]. Available: https://docs.kernel.org/driver-api/vfio.html (Accessed 2023-04-09).

[67] N. Jia and K. Wankhede. (2016) Vfio mediated devices. The kernel development community, NVIDIA CORPORATION. [Online]. Available: https://docs.kernel.org/driver-api/vfio-mediated-device.html (Accessed 2023-04-09).

[68] C. Schwarz, V. Reusch, and M. Planeta, "Dma security in the presence of iommus," in *Tagungsband des FG-BS Frühjahrstreffens 2022*. Bonn: Gesellschaft für Informatik e.V., 2022.

[69] Y. Zhang. (2017) Live migration with mdev device. Intel Open Source Technology Center. [Online]. Available: https://www.linux-kvm.org/images/f/fa/Live_migration_with_mdev_device_-_2017_0.pdf (Accessed 2023-04-09).

[70] K. Tian. (2017) Hardware-assisted mediated passthrough with vfio. Intel Open Source Technology Center. [Online]. Available: https://events19.linuxfoundation.org/wp-content/uploads/2017/12/Hardware-Assisted-Mediated-Pass-Through-with-VFIO-Kevin-pdf (Accessed 2023-04-09).

[71] M. Peresini, "I/o virtualization in networking," 2020. [Online]. Available: https://theses.cz/id/p8kd56/22465_Archive.pdf (Accessed 2023-04-09).

[72] B. Peng, H. Zhang, J. Yao, Y. Dong, Y. Xu, and H. Guan, "Mdev-nvme: A nvme storage virtualization solution with mediated pass-through," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 665–676.

[73] B. Peng, J. Yao, Y. Dong, and H. Guan, "Mdev-nvme: Mediated pass-through nvme virtualization solution with adaptive polling," *IEEE Transactions on Computers*, vol. 71, no. 2, pp. 251–265, 2022.

[74] C. Liang. (2018) Vdpa: Vhost-mdev as new vhost protocol transport. [Online]. Available: https://events19.linuxfoundation.org/wp-content/uploads/2017/12/Cunming-Liang-Intel-KVM-Forum-2018-VDPA-VHOST-MDE pdf (Accessed 2023-04-09).

[75] I. Corporation, *Intel® Ethernet Controller E810 Datasheet*, 2022. [Online]. Available: https://cdrdv2-public.intel.com/613875/613875_E810_Datasheet_Rev2.5.pdf (Accessed 2023-04-17).

[76] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the 2015*

Internet Measurement Conference, ser. IMC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 275–287. [Online]. Available: https://www.net.in.tum.de/fileadmin/bibtex/publications/papers/MoonGen_IMC2015.pdf (Accessed 2022-08-21).

[77] A. Beifuß, D. Raumer, P. Emmerich, T. M. Runge, F. Wohlfart, B. E. Wolfinger, and G. Carle, "A study of networking software induced latency," in *2015 International Conference and Workshops on Networked Systems (NetSys)*, 2015, pp. 1–8.

[78] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "A study of network stack latency for game servers," in *2014 13th Annual Workshop on Network and Systems Support for Games*, 2014, pp. 1–6.

[79] J. C. Mogul and L. Popa, "What we talk about when we talk about cloud network performance," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 5, p. 44–48, sep 2012. [Online]. Available: https://doi.org/10.1145/2378956.2378964

[80] R. Shea, F. Wang, H. Wang, and J. Liu, "A deep investigation into network performance in virtual machine based cloud environments," in *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, 2014, pp. 1285–1293.

[81] J. kang DONG, H. bo WANG, Y. yang LI, and S. duan CHENG, "Virtual machine placement optimizing to improve network performance in cloud data centers," *The Journal of China Universities of Posts and Telecommunications*, vol. 21, no. 3, pp. 62–70, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1005888514603022

[82] V. Persico, A. Montieri, and A. Pescapè, "On the network performance of amazon s3 cloud-storage service," in *2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*, Oct 2016, pp. 113–118.

[83] A. Vogel, D. Griebler, C. Schepke, and L. G. Fernandes, "An intra-cloud networking performance evaluation on cloudstack environment," in *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, March 2017, pp. 468–472.

[84] V. Persico, P. Marchetta, A. Botta, and A. Pescape, "On network throughput variability in microsoft azure cloud," in *2015 IEEE Global Communications Conference (GLOBECOM)*, Dec 2015, pp. 1–6.

[85] I. Pelle, J. Czentye, J. Dóka, and B. Sonkoly, "Towards latency sensitive cloud native applications: A performance study on aws," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, July 2019, pp. 272–280.

[86] F. Palumbo, G. Aceto, A. Botta, D. Ciuonzo, V. Persico, and A. Pescapé, "Characterization and analysis of cloud-to-user latency: The case of azure and aws," *Computer Networks*, vol. 184, p. 107693, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128620312962

[87] A. Gandhi and J. Chan, "Analyzing the network for aws distributed cloud computing," *SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 3, p. 12–15, nov 2015. [Online]. Available: https://doi.org/10.1145/2847220.2847224

[88] DPDK Developers, *DPDK Documentation*, 2022. [Online]. Available: https://core.dpdk.org/doc/ (Accessed 2022-08-30).

[89] Emmerich, Paul. (2022) vmuxio/vmuxio/test/moonprogs/l2-load-latency.lua. [Online]. Available: https://github.com/emmericp/MoonGen/blob/7746ff2f0afdbb222aa9cb220b48355e2d19552b/examples/l2-load-latency.lua (Accessed 2023-04-14).

[90] S.-A. Gierens. (2022) vmuxio/dat/bt/virtio. [Online]. Available: https://github.com/vmuxIO/dat/tree/ac3ca649928ebf75ab50b802f23d31a7370bd383/bt/virtio (Accessed 2023-04-07).

[91] S.-A. Gierens. (2022) vmuxio/plots/bt/virtio. [Online]. Available: https://github.com/vmuxIO/plots/blob/31dd9978fbfd380c5de075baff7627f7420ca02e/out/bt/virtio (Accessed 2023-04-07).

[92] J. Thalheim and P. Okelmann, *doctor-cluster-config/docs/hosts/river*, 2023. [Online]. Available: https://github.com/TUM-DSE/doctor-cluster-config/blob/0c70f0c95db6321d45e4182f84a753cf1eabadb3/docs/hosts/river.md (Accessed 2023-04-07).

[93] J. Thalheim and P. Okelmann, *doctor-cluster-config/docs/hosts/wilfred*, 2023. [Online]. Available: https://github.com/TUM-DSE/doctor-cluster-config/blob/0c70f0c95db6321d45e4182f84a753cf1eabadb3/docs/hosts/wilfred.md (Accessed 2023-04-07).

[94] S.-A. Gierens. (2022) vmuxio/dat/gr/virtio. [Online]. Available: https://github.com/vmuxIO/dat/blob/e2c4a1700e1b7c1881f9142710bb6c525d702b4f/gr/virtio2 (Accessed 2023-04-07).

[95] S.-A. Gierens. (2022) vmuxio/plots/gr/virtio. [Online]. Available: https://github.com/vmuxIO/plots/tree/7d9b582c2a94fd7394f63d5fdcc0f2c112263b5c/out/gr/virtio2 (Accessed 2023-04-07).

[96] Sandro-Alessio Gierens, The QEMU Project Developers. (2022) qemu ioregionfd: Add posted_writes argument to virtio_ioregionfd_init. [Online]. Available: https://github.com/vmuxIO/qemu/commit/8eee73f75751834395e88c165059e93f18c7a024 (Accessed 2023-04-14).

[97] I. D. V. team. (2020) Dpdk intel nic performance report release 20.11. [Online]. Available: https://fast.dpdk.org/doc/perf/DPDK_20_11_Intel_NIC_performance_report.pdf (Accessed 2021-04-15).

[98] Gierens, Sandro-Alessio and Emmerich, Paul. (2022) vmuxio/vmuxio/test/moonprogs/l2-load-latency.lua. [Online]. Available: https://github.com/vmuxIO/vmuxIO/blob/a99575e903de40c7dd2f39905498d59d2dab8985/test/moonprogs/l2-load-latency.lua (Accessed 2023-04-14).

[99] Emmerich, Paul and Bonk, Fabian. (2022) libmoon/libmoon/examples/l2-load-latency.lua. [Online]. Available: https://github.com/libmoon/libmoon/blob/2dbbcd909874fc84ceecb7dd82e45d4cd28cef50/examples/reflector.lua (Accessed 2023-04-14).

[100] X.-L. Wu, W.-M. Li, F. Liu, and H. Yu, "Packet size distribution of typical internet applications," in *2012 International Conference on Wavelet Active Media Technology and Information Processing (ICWAMTIP)*. IEEE, 2012, pp. 276–281.

[101] R. Sinha, C. Papadopoulos, and J. Heidemann, "Internet packet size distributions: Some observations," *USC/Information Sciences Institute, Tech. Rep. ISI-TR-2007-643*, pp. 1536–1276, 2007.

[102] E. Garsva, N. Paulauskas, and G. Grazulevicius, "Packet size distribution tendencies in computer network flows," in *2015 Open Conference of Electrical, Electronic and Information Sciences (eStream)*. IEEE, 2015, pp. 1–6.

[103] K. Pentikousis and H. Badr, "Quantifying the deployment of tcp options - a comparative study," *IEEE Communications Letters*, vol. 8, no. 10, pp. 647–649, 2004.

[104] D. Murray and T. Konziniec, "The state of enterprise network traffic in 2012," 2012. [Online]. Available: https://citeseerx.ist.psu.edu/doc/10.1.1.707.5199 (Accessed 2023-04-17).

[105] M. Zhang, M. Dusi, J. Wolfgang, and C. Chen, "Analysis of udp traffic usage on internet backbone links," 2009. [Online]. Available: https://citeseerx.ist.psu.edu/doc/10.1.1.534.6303 (Accessed 2023-04-17).

[106] W. John and S. Tafvelin, "Analysis of internet backbone traffic and header anomalies observed," 2007. [Online]. Available: http://conferences.sigcomm.org/imc/2007/papers/imc91.pdf (Accessed 2023-04-17).

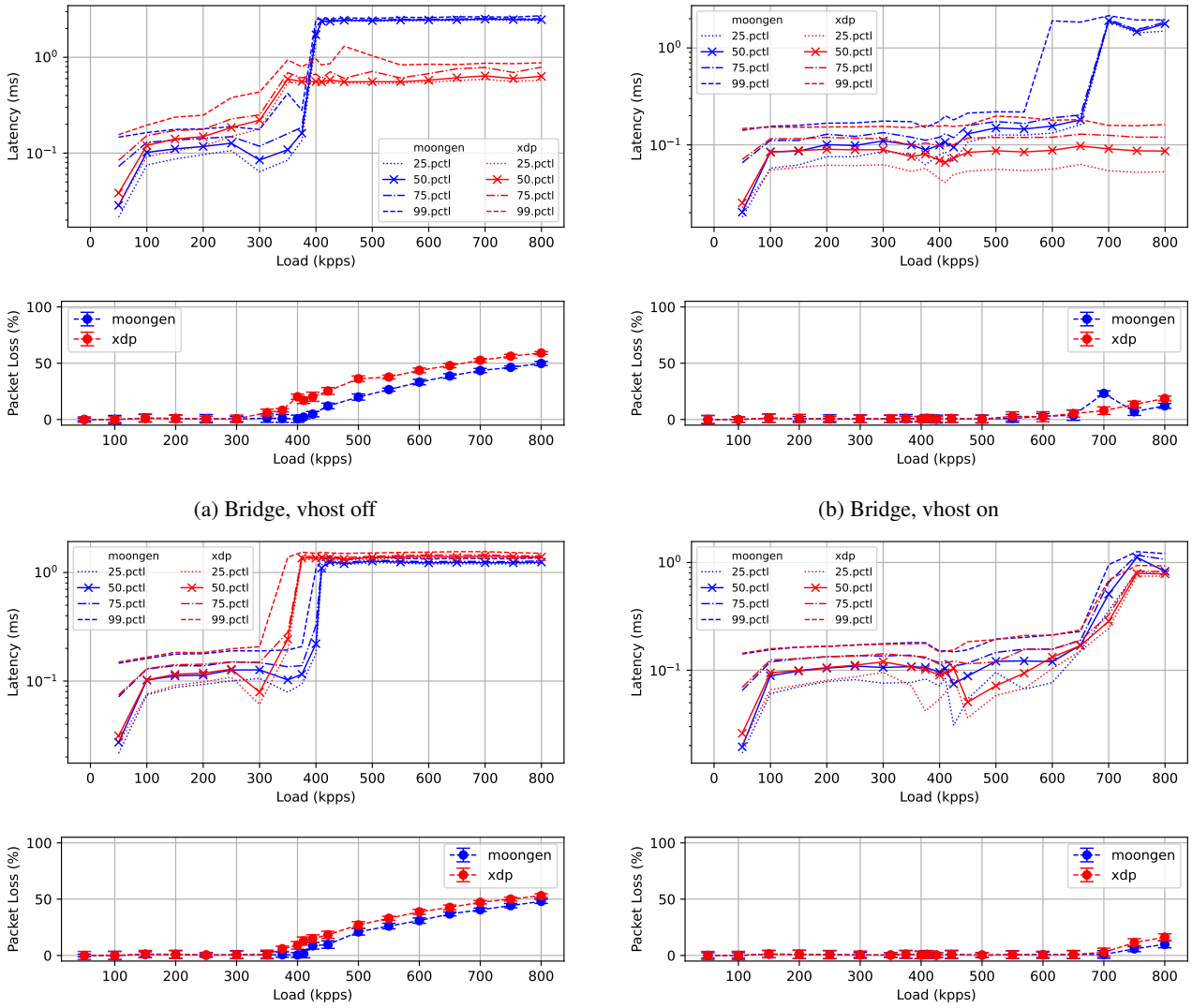
[107] "Test methodology journal," *IMIX (Internet Mix) Journal*, 2006. [Online]. Available: https://web.archive.org/web/20150223213659/http://www.spirent.com/~/media/Test%20Methodology%20Journal/Spirent_TestMethodology_IMIX-Journal.pdf (Accessed 2023-04-17).

[108] "The journal of internet test methodologies," 2007. [Online]. Available: https://web.archive.org/web/20130127153505/http://www.ixiacom.com/pdfs/test_plans/agilent_journal_of_internet_test_methodologies.pdf (Accessed 2023-04-17).

6 Appendix

6.1 Virtio Ioregionfd Comparison Plots

6.1.1 Reflector



(a) Bridge, vhost off

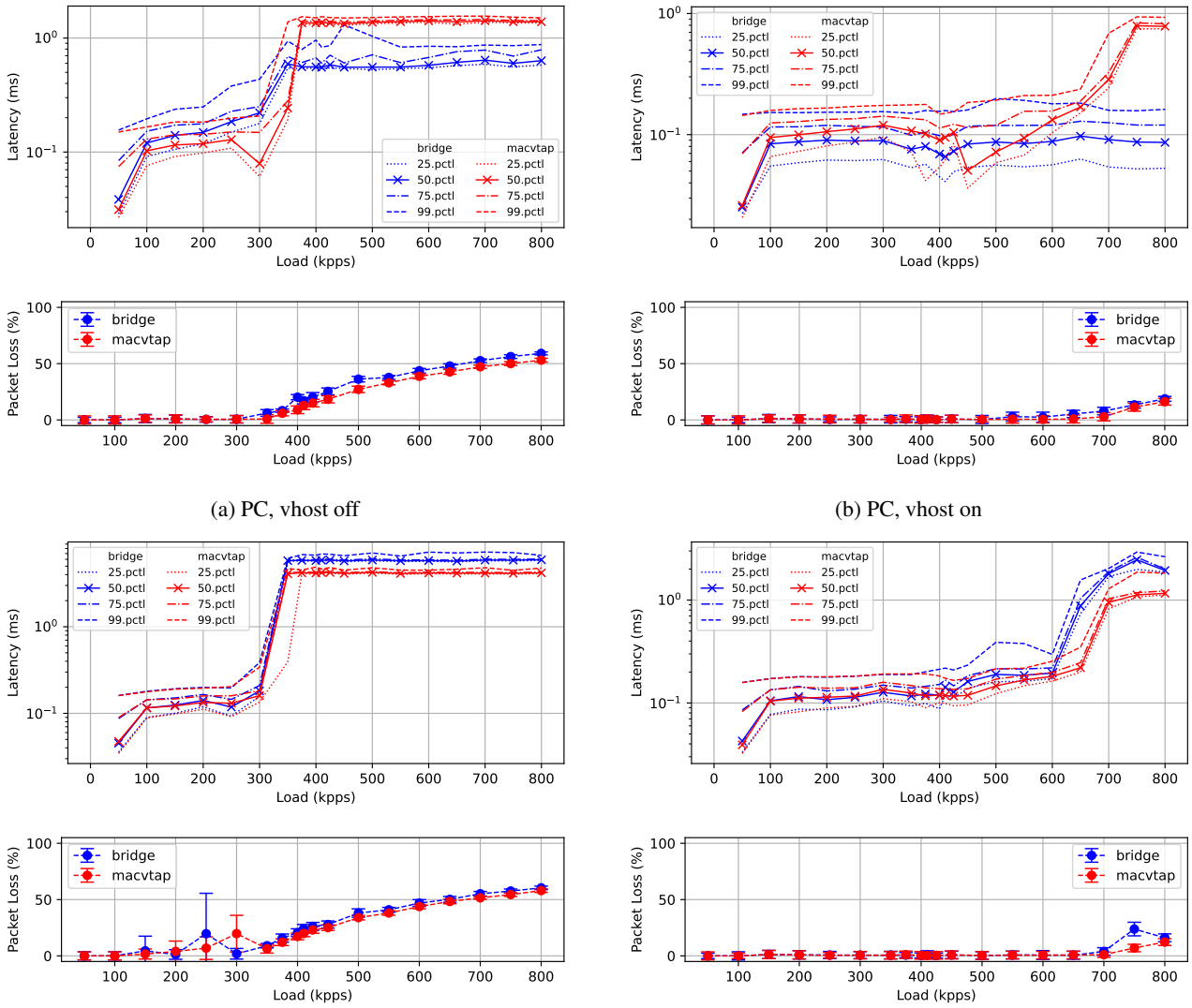
(b) Bridge, vhost on

(c) MacVTap, vhost off

(d) MacVTap, vhost on

Figure 9: Load latency and packet loss comparison for different reflectors (MoonGen vs. XDP), Bridge (top) and MacVTap (bottom), with vhost off (left) and vhost on (right). Packet size is 64 B, bus type is PCI, and ioregionfd is off.

6.1.2 Interface



(a) PC, vhost off

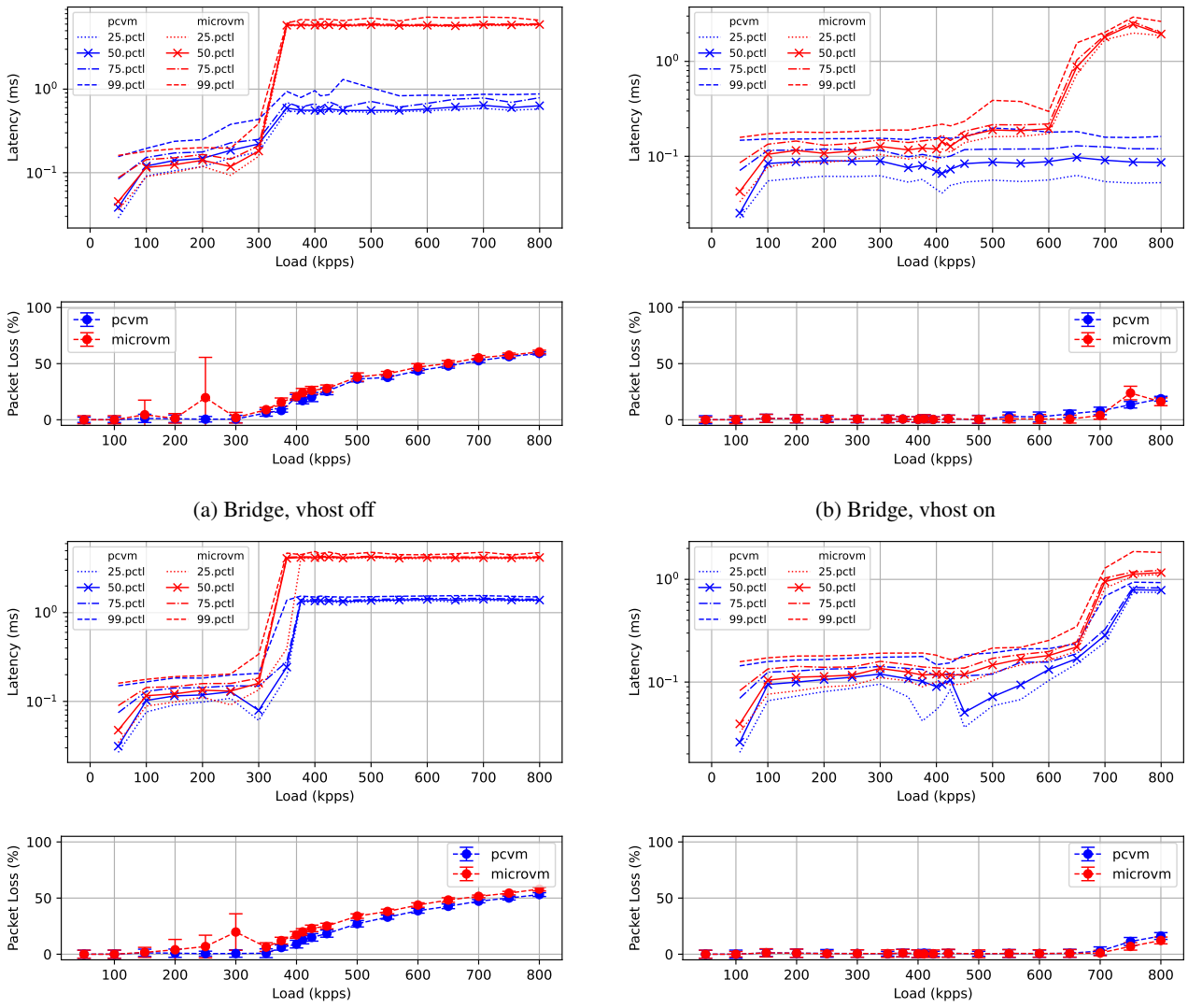
(b) PC, vhost on

(c) MicroVM, vhost off

(d) MicroVM, vhost on

Figure 10: Load latency and packet loss comparison for different interfaces (Bridge vs. MacVTap), PC (top) and MicroVM (bottom), with vhost off (left) and vhost on (right). Packet size is 64 B, reflector is XDP, and ioregionfd is off.

6.1.3 Bus Type



(a) Bridge, vhost off

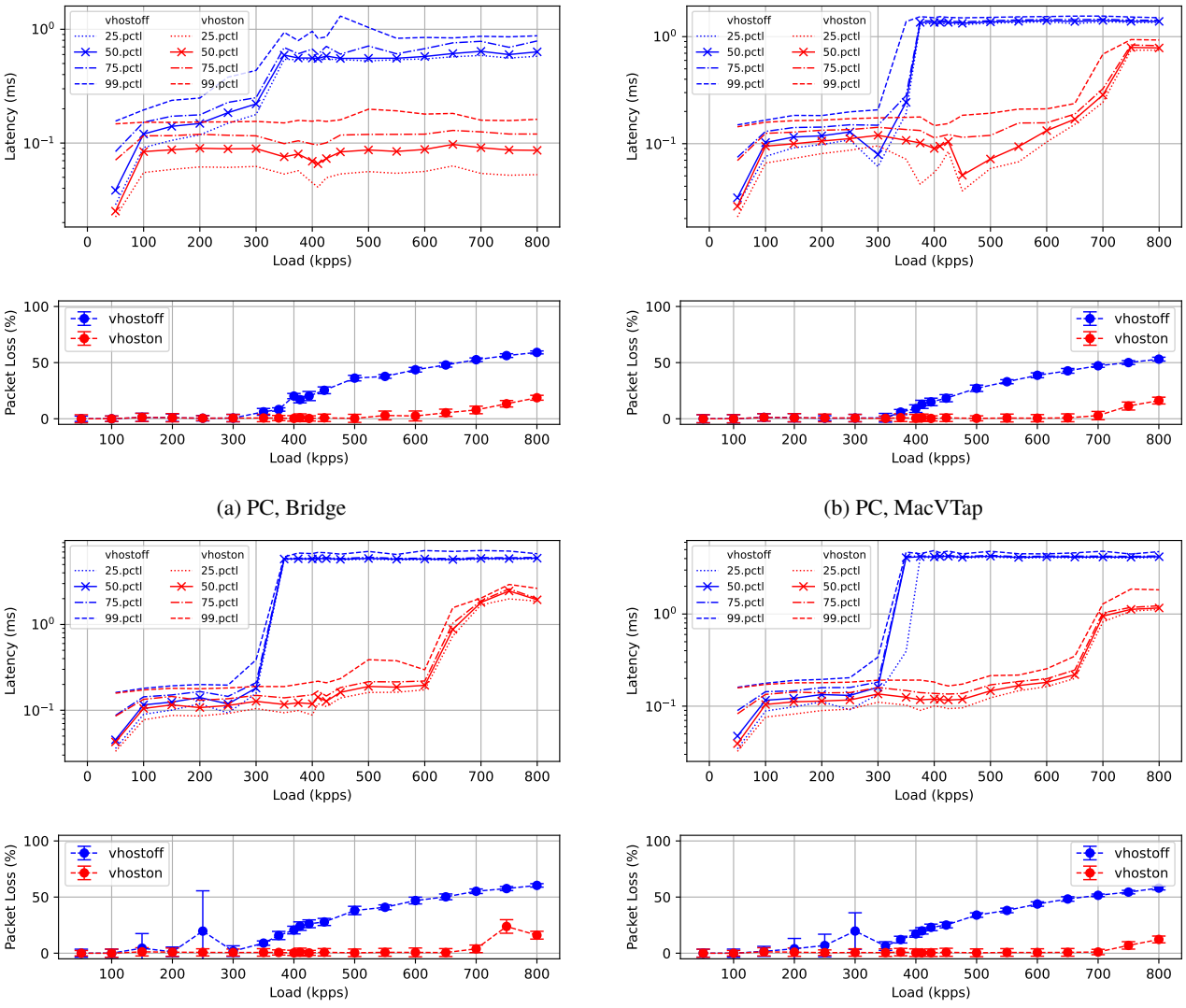
(b) Bridge, vhost on

(c) MacVTap, vhost off

(d) MacVTap, vhost on

Figure 11: Load latency and packet loss comparison for different machine types (pc vs. microvm), and therefore different bus types (PCI vs. MMIO), thus between virtio-net-pci (blue) and virtio-net-device (red), Bridge (top) and MacVTap (bottom), with vhost off (left) and vhost on (right). Packet size is 64 B, XDP is used as reflector, and ioregionfd is off.

6.1.4 Vhost Protocol



(a) PC, Bridge

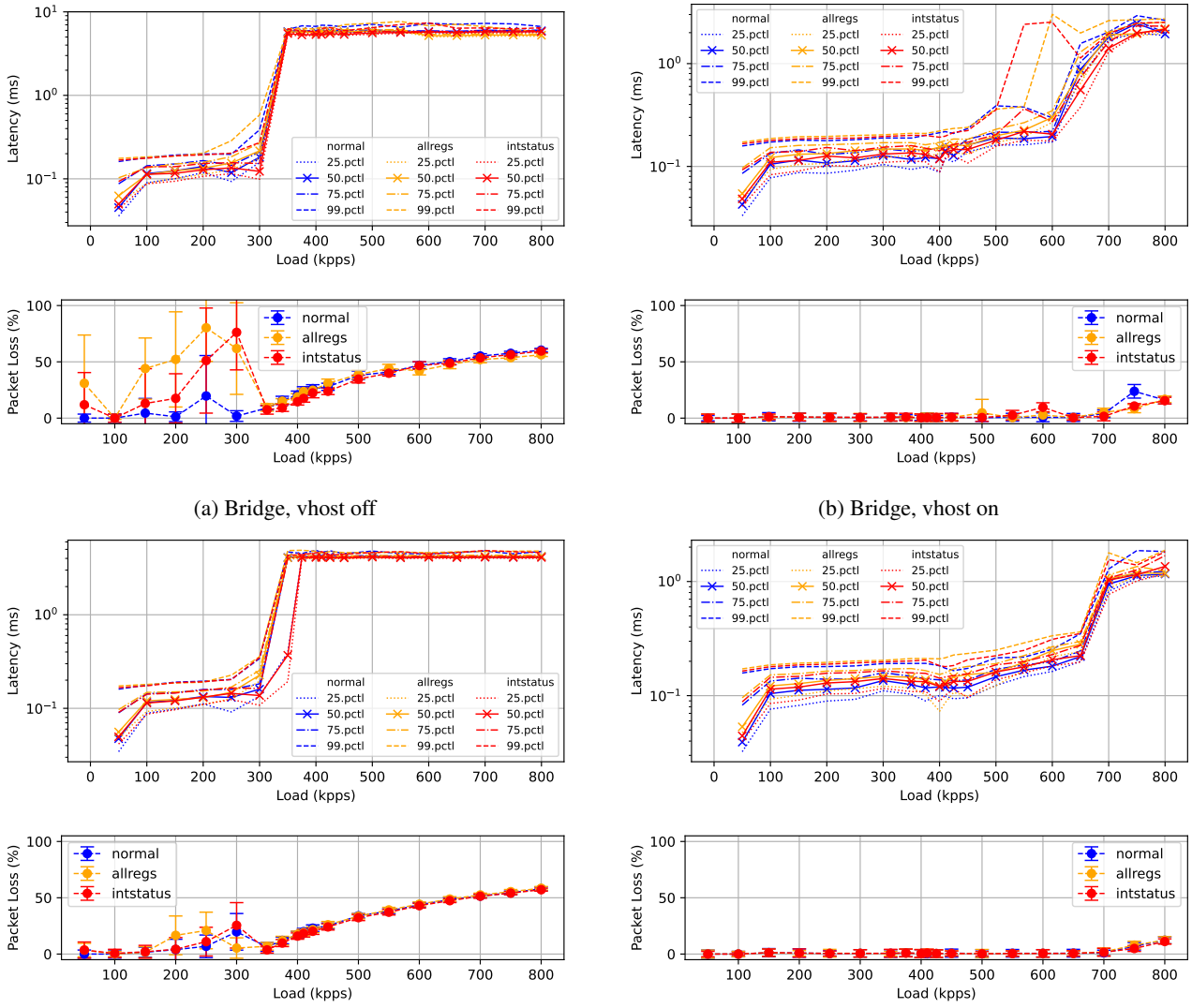
(b) PC, MacVTap

(c) MicroVM, Bridge

(d) MicroVM, MacVTap

Figure 12: Load latency and packet loss comparison for different vhost settings, PC (top) and MicroVM (bottom), Bridge (left) and MacVTap (right). Packet size is 64 B, reflector is XDP, and ioregionfd is off.

6.1.5 Ioregionfd



(a) Bridge, vhost off

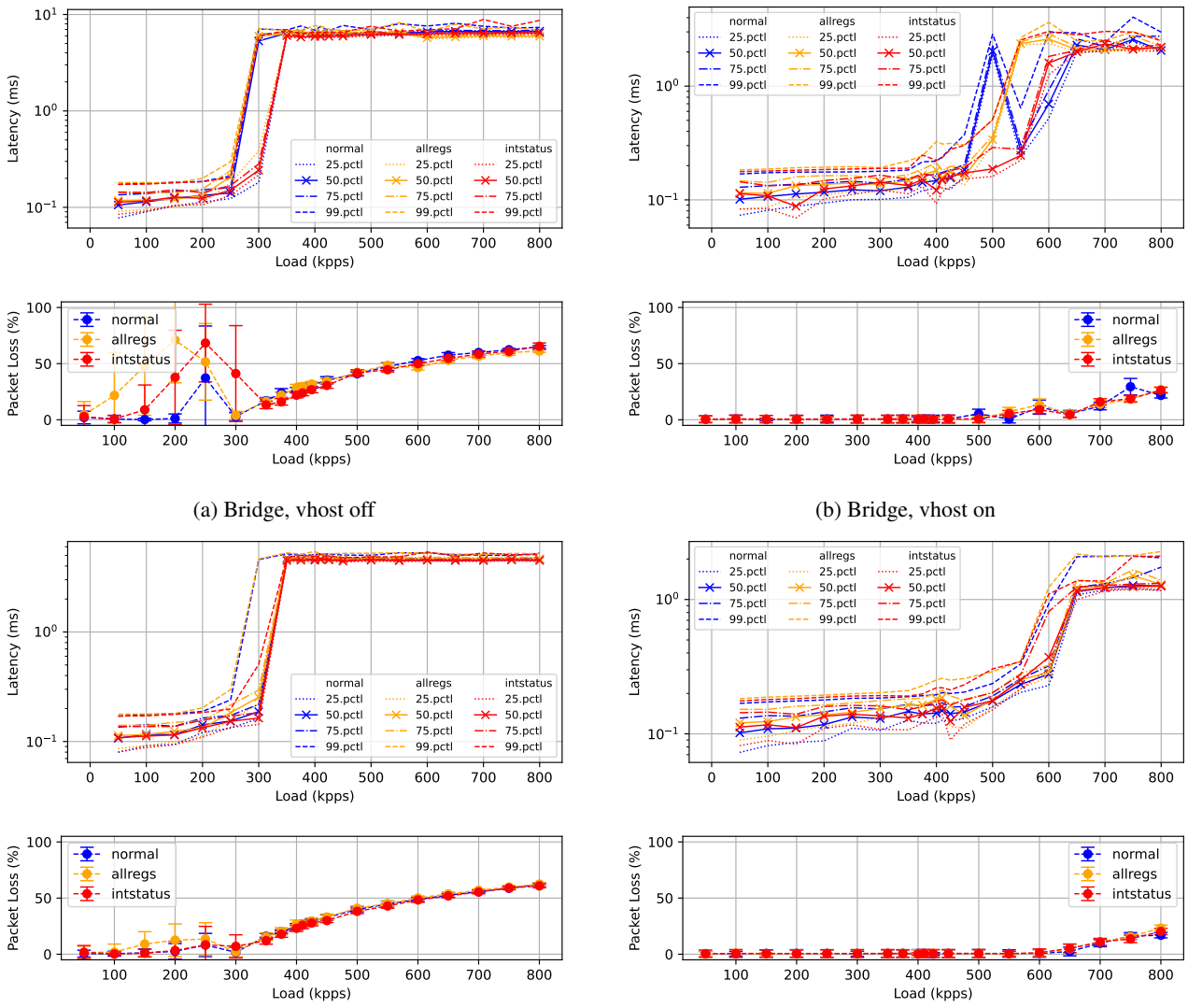
(b) Bridge, vhost on

(c) MacVTap, vhost off

(d) MacVTap, vhost on

Figure 13: Load latency and packet loss comparison for ioregionfd off or on (on the interrupt status register (intstatus) or all registers (allregs) that work with ioregionfd), Bridge (top) and MacVTap (bottom), with vhost off (left) and vhost on (right). Packet size is 64 B, bus type is MMIO, and reflector is XDP.

6.1.6 Ioregionfd with Large Packets



(a) Bridge, vhost off

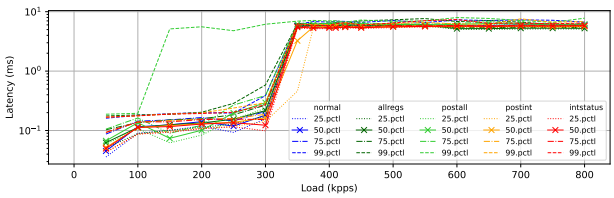
(b) Bridge, vhost on

(c) MacVTap, vhost off

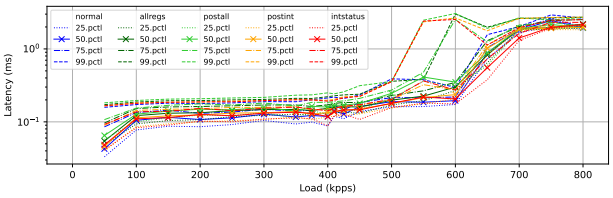
(d) MacVTap, vhost on

Figure 14: Load latency and packet loss comparison for ioregionfd off or on (on the interrupt status register (intstatus) or all registers (allregs) that work with ioregionfd), Bridge (top) and MacVTap (bottom), with vhost off (left) and vhost on (right). Packet size is 1024 B, bus type is MMIO, and reflector is XDP.

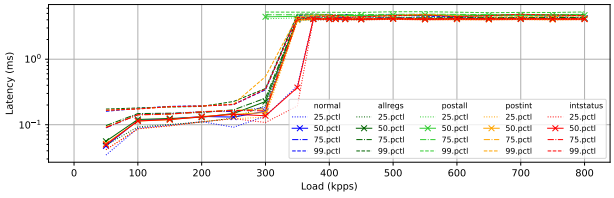
6.1.7 Ioregionfd Posted Writes



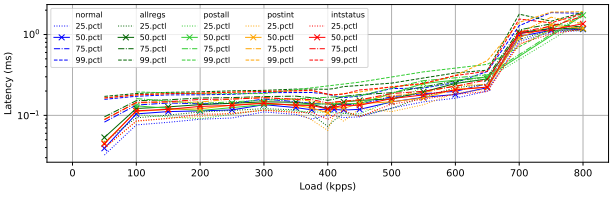
(a) Bridge, vhost off



(b) Bridge, vhost on



(c) MacVTap, vhost off



(d) MacVTap, vhost on

Figure 15: Load latency and packet loss comparison for ioregionfd off or on (on the interrupt status register (intstatus) or all registers (allregs) that work with ioregionfd) and with posted writes (on interrupt status (postint) and all registers (postall)), Bridge (top) and MacVTap (bottom), with vhost off (left) and vhost on (right). Packet size is 64 B, bus type is MMIO, and reflector is XDP.