Performance Impact of the IOMMU for DPDK

Marcel Gaupp, Sebastian Gallenmüller*, Stefan Lachnit*
*Chair of Network Architectures and Services
School of Computation, Information and Technology, Technical University of Munich, Germany
Email: m.gaupp@tum.de, gallenmu@net.in.tum.de, lachnit@net.in.tum.de

***Abstract*—The DPDK framework is used in a range of industries where high-performance packet processing is required. One of the available DPDK drivers is based on the VFIO Linux API, which makes use of the system's IOMMU. The VFIO based drivers are particularly useful for virtual machines.**

Previous research suggested that the IOMMU can have significant performance impacts on I/O operations. This paper measures the performance differences between a VFIO driver and an UIO driver, which does not use the IOMMU. We measure throughput and latency of NICs in a reproducible testing setup.

The results show that the performance difference is not significant in the average case and that a few packets show higher latency. This is explained by the caching behavior of the TLB.

***Index Terms*—computer networks, system buses, measurement, high-speed networks**

1. Introduction

The Data Plane Development Kit (DPDK) enables the development of various high-performance network applications. It is used in data centers, the network edge and infrastructure systems where data processing is performance critical.

Virtual machines (VMs) are often used to improve scalability and security. The secure and efficient use of network interface controllers (NICs) within VMs require special consideration. The use of the VFIO Linux API enables the secure assignment of devices to virtual machines.

In this paper, we compare the performance of two DPDK drivers: An older UIO based driver and the newer VFIO based driver. The VFIO driver mainly differs from the UIO driver in its use of the I/O Memory Management Unit (IOMMU). We show how the IOMMU affects the throughput and latency performance of NICs when used with DPDK.

This paper is structured as follows: Chapter 2 presents related work this paper is based on and differentiates our work from theirs. Chapter 3 introduces background information on PCIe, DPDK, and the IOMMU required to understand the analysis. Chapter 4 analyses differences of the different drivers and how these differences might affect performance. Chapter 5 describes the measurement setup and methodology. Chapter 6 presents and interprets our results and explains how they could have come to be. Chapter 7 concludes with a summary of our findings and their implications.

2. Related work

This paper is based on the findings of Neugebauer et al. [1] which show a significant drop in bandwidth if an IOMMU is used with low packet sizes. Their experiments differ from ours in that they allocated memory buffers sequentially within varying ranges in host memory. In our experiments, we relied on MoonGen's memory allocation strategy, which is not sequential.

Furthermore, Neugebauer et al. did not apply their research on DPDK, but on the use of PCIe in general. That is why they decided not to use the hugepages feature of Linux which increases the page size. Hugepage support is required by DPDK, which is why we did enable hugepages.

3. Background

Several key components contribute to the resulting performance analysis. This section provides an overview of the foundational knowledge about PCIe, DPDK and the IOMMU.

3.1. PCIe

The Peripheral Component Interconnect Express (PCIe) is a communication and interconnect standard interface [2]. It is the de-facto interface for connecting peripheral I/O devices, like NICs, to x86 based computers [1].

It uses a packeted communication protocol to send data between devices. Direct Memory Accesses (DMAs) are data transfers between the device and the host memory. In PCIe, they are implemented by Memory Read and Memory Write packets.

3.2. DPDK

DPDK is a set of libraries which provide a framework for creating network applications [3]. One use case is the creation of poll mode drivers (PMDs). PMDs can achieve higher performance in bandwidth and latency than their interrupt based counterparts, because they disable interrupts and poll write-back descriptors in host memory, thus leaving more bandwidth on the PCI bus for packet data [1].

 doi: 10.2313/NET-2024-09-1_11

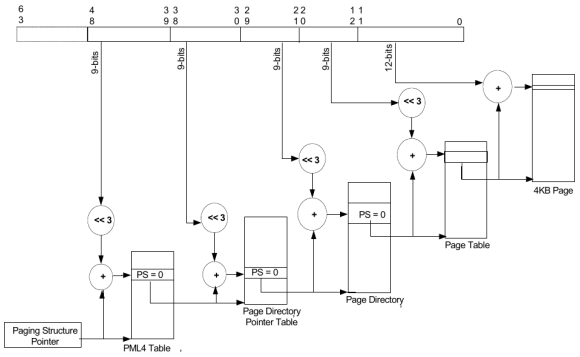

Figure 1: Page Table Walk, taken from [6]

3.3. IOMMU

Newer systems include an IOMMU. If enabled, every DMA passes through the IOMMU. The IOMMU interprets the memory address as an I/O Virtual Address (IOVA) and translates it into a physical address [4]. This is similar to how an MMU translates virtual addresses of a process on the CPU into physical addresses.

Without an IOMMU, giving a virtual machine access to a DMA capable device would give the VM access to all of the host memory, because the DMAs can access all physical addresses [5]. This poses a threat to the security and robustness of the system as memory locations not belonging to the VM can be read and overwritten. Thus hypervisors have to emulate the device with a managed memory space. The hypervisor intercepts the real device's DMA and copies the data to the VM's memory space. This indirection has a detrimental effect on performance.

The IOMMU isolates the IOVA space from the physical address space, therefore restricting the devices to the configured memory pages. Like with MMUs, the mapping is configured in page tables. Multiple memory locations need to be accessed, as shown in Figure 1 to find the page of an IOVA [6]. The results of this page table walk are cached in the Translation Lookaside Buffer (TLB), which is where subsequent address translations of recently looked up pages are found in.

4. Analysis

The Linux kernel currently provides 2 interfaces for accessing IO memory from userspace: Userspace I/O (UIO) and Virtual Function I/O (VFIO).

Both types of drivers provide an interface which allows to map the device's memory ranges to the address space of an userspace process.

UIO requires kernel code, which initializes the device, defines device memory ranges to be mapped and potentially registers an interrupt handler [7]. Once its driver is bound to the device, UIO provides a device file located at `/dev/uioX`. Its file descriptor can be used with a call to `mmap()` to map the defined memory ranges to userspace.

VFIO extends this by allowing the creation of Virtual Functions and by allowing configuration of the IOMMU [8]. Virtual Functions are virtual copies of the device, which can be assigned to multiple virtual machines. This enables the sharing of the physical NIC in multiple VMs with almost no overhead.

TABLE 1: Node Configuration

Component	Description
CPU	Intel(R) Xeon(R) CPU D-1518 @ 2.20GHz
Microarchitecture	Broadwell
Memory	32 GiB
OS	Debian 12
Kernel	Linux 6.1.0-17
NIC	Intel X552 10 GbE SFP+

The configuration granularity of VFIO is that of an IOMMU group [9]. An IOMMU group is a group of devices which can be isolated from the rest of the system. How many and which devices are in such a group depends on the system topology. Once all devices are bound to the VFIO driver, the device file `/dev/vfio/`*`group`* can be used with calls to `ioctl()` and appropriate arguments to access device memory and to configure the IOMMU.

The IOMMU enables virtual addressing for DMAs. The virtual address of each access is translated by the IOMMU to the corresponding physical address. The translation is cached inside the TLB. On a TLB-miss, the page table, which resides in host memory, has to be walked. Because this page walk needs to access multiple memory locations, we theorize that page misses could incur significant performance impacts.

Since the UIO drivers do not use the IOMMU while the VFIO drivers do, one would expect to measure differences in throughput and latency.

5. Implementation

We conducted our experiments on a three node setup depicted in Figure 2. All nodes are identically configured as shown in Table 1. The optical splitters allow the node named `bitcoingold` to passively listen into the communication between `bitcoin` and `bitcoincash`.

For deploying and running our scripts on these hosts, we used the Plain Orchestrating Service (*pos*) [10]. It is a framework, which allows full automation of orchestration, measurement and evaluation with the goal to make the experiments easily reproducible. Our experimentation scripts[1] can be executed on a *pos* testbed controller connected to the three nodes configured as described before.

For packet generation and measurement we used MoonGen [11]. MoonGen is a Lua wrapper for DPDK and will, as such, use the DPDK driver bound to the NIC. We ran the MoonGen tasks with 4 threads where applicable, so that no operation would be CPU bound.

The drivers we compare and their configuration are listed in Table 2. While other options like `uio_pci_generic` and `vfio enable_unsafe_noiommu_mode=1` exist [12], our chosen drivers have been available for the longest time. Therefore, they particularly are of interest for legacy applications.

5.1. Throughput

For the throughput measurement, we used the `bitcoin` and `bitcoincash` nodes and ignored the `bitcoingold`

1. https://gitlab.lrz.de/marcel.gaupp/dpdk-iommu-effects

 doi: 10.2313/NET-2024-09-1_11

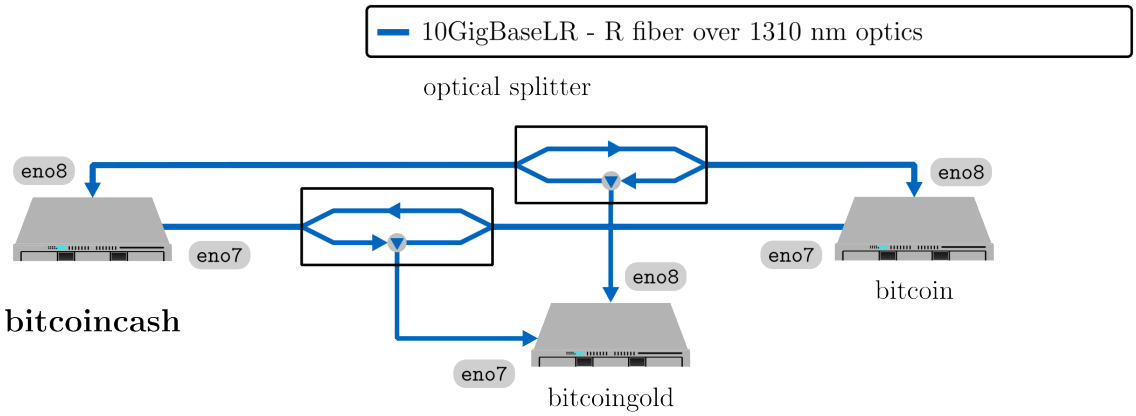

Figure 2: Topology

TABLE 2: Driver Configuration

Kernel Module	Boot Parameters
igb_uio	intel_iommu=off iommu=pt
vfio-pci	intel_iommu=on

host. One host was configured as a load generator and the other was configured as a Layer-2-Forwarder. The load generator generated packets as fast as possible and the forwarder returned them to the sender. Both nodes only used one interface port (eno7) to communicate.

The load generator sent packets from sizes 48 Bytes to 88 Bytes in increments of 4 Bytes. The packets were sent for 60 s for each packet size. igb_uio was always selected as the driver, assuming as a UIO driver it would be at least as fast as the other option.

This command was run on the load generator:

```
/root/moongen/build/MoonGen
  /root/code/pktgen.lua 0
  -s "$PACKET_SIZE" --threads 4
```

The pktgen.lua script sends UDP packets to some nonexistent destination on the given DPDK port id (0). $PACKET_SIZE was a *pos* loop variable, which differed for each experiment run.

The measurements were made on the forwarder, which was the device under test (DUT). We measured ingress and egress throughput for each driver configuration. This command was run on the DUT:

```
/root/moongen/build/MoonGen
  /root/code/l2-forward.lua 0 0
```

The l2-forward.lua script simply forwards every packet received on the first port to the second port (both 0). It also produces an average throughput statistic every second.

5.2. Latency

For the latency measurement, once again, the bitcoin and bitcoincash nodes were configured as load generator and forwarder, with the forwarder being the DUT, which is tested with both drivers. But this time the forwarded response was sent back on the other interface port:

```
/root/moongen/build/MoonGen
  /root/code/l2-forward.lua 1 0
```

The bitcoingold node was configured to timestamp the packets in both directions. The NIC supports hardware timestamping and achieves an accuracy of below 100 ns [11]. The time difference between the same packet being received by the DUT and the same packed being transmitted by the DUT was recorded as the latency. The packets were identified by a unique identifier in the UDP payload. To generate these packets, this was run on the load generator:

```
/root/moongen/build/MoonGen
  /root/code/traffic-gen.lua 1 0
  -t "$DURATION" -s $PACKET_SIZE -r $RATE
```

The traffic-gen.lua script generates UDP packets with an increasing 32-bit integer as the payload. Packets were sent on interface port 1, while port 0 received packets for statistics. The transmission rate was fixed at 1 Gbit/s and the duration was 60 s.

This time we varied the packet size starting at 64 Bytes, doubling until the MTU of 1500 Bytes was reached.

To measure the latency, the capturing host (bitcoingold) ran this command:

```
/root/moongen/build/MoonGen
  /root/code/sniffer.lua 1 0
  -t 300 --seq-offset 42
```

This script captures and timestamps the packets on interface ports 1 and 0 and records their timestamps and identifiers in the latencies-pre.mscap and latencies-post.mscap respectively. The runtime -t 300 is set to 300 s, much longer than the packet generation time of 60 s. But the script is killed once

 doi: 10.2313/NET-2024-09-1_11

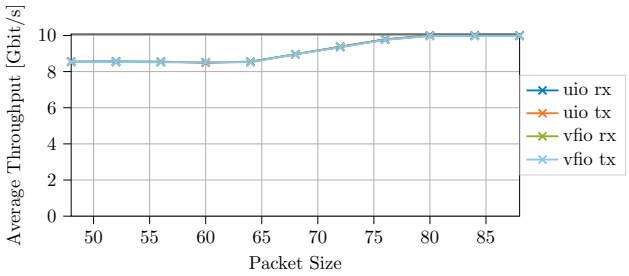

Figure 3: Throughput

the *pos* framework detects that the packet generator has stopped running. The `--seq-offset` option specifies the offset of the identifier from the start of the Ethernet frame. We calculated this with the Ethernet header being 14 Bytes, the IPv4 header being 20 Bytes and the UDP header being 8 Bytes: $14 + 20 + 8 = 42$

Afterwards, we processed these records to generate histogram data:

```
/root/moongen/build/MoonGen
  /root/moongen/examples/
      moonsniff/post-processing.lua
  -i latencies-pre.mscap -s latencies-post.mscap
```

This generates the `hist.csv`, which pairs latencies in nanoseconds with their occurrence count.

6. Evaluation

The results show that the average performance impact is not as significant as we expected.

6.1. Throughput

Figure 3 shows the average ingress and egress throughput of both driver variants. However, all variants are similar enough such that no difference is visible.

For sizes 48 B to 64 B, the throughput is constant but slightly lower than the capacity of the NIC. It increases from 64 B to 80 B where it reaches the full 10 Gbit/s.

The lower throughput for packet sizes below 80 B can be explained by the packetized structure of the PCIe protocol [1]. For smaller network packet sizes the size of the PCIe level packets are dominated by the PCIe packet headers. Therefore, less data is transmitted while the PCIe bus bandwidth is used up.

For packet sizes below 64 B our theory is that the hardware pads the size up to 64 B because this is the minimum Ethernet frame size.

6.2. Latency

The latency's percentiles below 99 show very little variation as show in Figure 4. We notice a slight linear increase of the latency for increasing packet sizes.

If we look at the higher percentiles shown in Figure 5, significant differences between the drivers become visible. This means that something has an effect on a few packets and this effect differs for the drivers.

We theorize that the TLB does not get completely filled up and that only the first packet of the corresponding

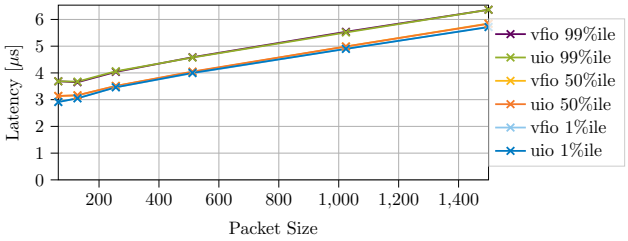

Figure 4: Latency middle percentiles

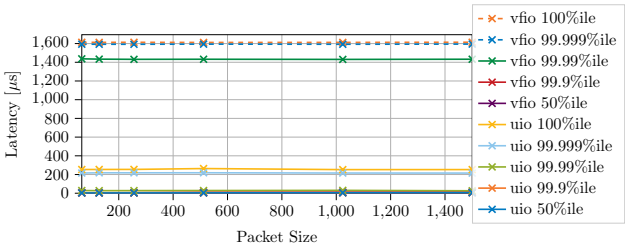

Figure 5: Latency upper percentiles

page causes a page walk to happen. This explains how only a few packets (the packets causing a page walk) do have a measurable difference.

This result differs from those of Neugebauer et al. [1] because they allocated DMA buffers linearly, while we used MoonGen's allocations which reuse buffers. A linear allocation strategy covers a wider range of memory addresses and uses more memory pages. Because fewer memory pages are used this results in the TLB not filling up.

Furthermore, we were forced to use hugepages, which increases the page size from 4 KiB to 2 MiB. Not only does this reduce the page count per memory used, it also increases the size of the page tables. This significantly reduces how often a full page walk has to be executed.

7. Conclusion

This paper answers whether the IOMMU does have a performance impact if used with DPDK. We show that the `vfio-pci` driver for DPDK and its use of the IOMMU do not have a significant performance impact for regular memory access patterns. Only the first few accesses show increased access time. Further accesses of cached pages show no measurable delay if accessed via an IOMMU.

VFIO drivers do not come at a performance cost. That is why we believe that the use of them will make systems more reliable and virtual machines more efficient securely without compromise.

References

[1] 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*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 327–341. [Online]. Available: https://doi.org/10.1145/3230543.3230560

[2] H. Zhu, *Data Plane Development Kit (DPDK): A Software Optimization Guide to the User Space-Based Network Applications*. CRC Press, 2020.

[3] "About dpdk," 2024, accessed 15. June 2024. [Online]. Available: https://www.dpdk.org/about/

[4] O. Peleg, A. Morrison, B. Serebrin, and D. Tsafrir, "Utilizing the IOMMU scalably," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, Jul. 2015, pp. 549–562. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/peleg

[5] R. Jithin and P. Chandran, "Virtual machine isolation," in *Recent Trends in Computer Networks and Distributed Systems Security*, G. Martínez Pérez, S. M. Thampi, R. Ko, and L. Shu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 91–102.

[6] *Intel Virtualization Technology for Directed I/O Architecture Specification*, 4th ed., Intel Corporation, June 2022.

[7] J. Corbet, "Uio: user-space drivers," *LWN.net*, 2007, accessed 16. June 2024. [Online]. Available: https://lwn.net/Articles/232575/

[8] ——, "Safe device assignment with vfio," *LWN.net*, 2012, accessed 12. June 2024. [Online]. Available: https://lwn.net/Articles/474088/

[9] *The Linux Kernel documentation / VFIO - "Virtual Function I/O"*, 2024, accessed 16. June 2024. [Online]. Available: https://docs.kernel.org/driver-api/vfio.html

[10] S. Gallenmüller, D. Scholz, H. Stubbe, and G. Carle, "The pos framework: a methodology and toolchain for reproducible network experiments," in *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 259–266. [Online]. Available: https://doi.org/10.1145/3485983.3494841

[11] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," in *Internet Measurement Conference 2015 (IMC'15)*, Tokyo, Japan, Oct. 2015.

[12] *DPDK documentation / Linux Drivers*, 2024, accessed 12. June 2024. [Online]. Available: https://doc.dpdk.org/guides-24.03/linux_gsg/linux_drivers.html