HAVS: Hardware-accelerated Shared-memory-based VPP Network Stack

Shujun Zhuang*, Jian Zhao*, Jian Li*§, Ping Yu‡, Yuwei Zhang‡ and Haibing Guan†
*School of Software Engineering, †Department of Computer Science and Engineering,
Shanghai Jiao Tong University, Shanghai, China
‡Intel Asia-Pacific R&D Ltd.

IEEE INFOCOM 2021 - IEEE Conference on Computer Communications | 978-1-6654-0325-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/INFOCOM42981.2021.9488808

***Abstract*—The number of requests to transfer large files is increasing rapidly in web server and remote-storage scenarios, and this increase requires a higher processing capacity from the network stack. However, to fully decouple from applications, many latest userspace network stacks, such as VPP (vector packet processing) and snap, adopt a shared-memory-based solution to communicate with upper applications. During this communication, the application or network stack needs to copy data to or from shared memory queues. In our verification experiment, these multiple copy operations incur more than 50% CPU consumption and severe performance degradation when the transferred file is larger than 32 KB.**

This paper adopts a hardware-accelerated solution and proposes HAVS which integrates Intel I/O Acceleration Technology into the VPP network stack to achieve high-performance memory copy offloading. An asynchronous copy architecture is introduced in HAVS to free up CPU resources. Moreover, an abstract memcpy accelerator layer is constructed in HAVS to ease the use of different types of hardware accelerators and sustain high availability with a fault-tolerance mechanism. The comprehensive evaluation shows that HAVS can provide an average 50%-60% throughput improvement over the original VPP stack when accelerating the nginx and SPDK iSCSI target application.

I. Introduction

Transferring large files over the network has become increasingly common [1]. With the rapid increase in webpage sizes, web server applications transfer webpages larger than 1 MB [2]. Moreover, a number of requests for reading and writing large files have emerged for remote-storage services in the cloud. This ever-increasing requested size for web server and remote-storage scenarios demands a higher processing capacity of the network stack.

With the increasing popularity of the userspace network stack [3]–[6], different approaches for providing network services to upper network applications, such as nginx [7], have been adopted. Referring to the classification of these approaches in snap [6], LibOS [8]–[10] and shared-memory-based approaches [5], [6] are two prevalent solutions. The LibOS approach puts network functionality into uncoordinated application libraries and communicates via a function call while the shared-memory-based approach starts the network stack as a separate host service and communicates with applications via lock-free shared-memory queues. Compared with the LibOS approach, the shared-memory-based approach adopted by VPP and snap has weak coupling and transparency advantages, so many useful features, including in-service upgrades, centralized resource accounting and enhanced security, can be easily implemented [6]. However, this approach also introduces extra memcpy operations between the network stack and applications compared with LibOS approach, and these copy operations become the main performance bottleneck when transferring large files. In our verification experiment, these copy operations consume more than 50% of CPU resources when the transferred file is larger than 32-KB. This paper targets this performance bottleneck in the shared-memory-based network stack and performs a series of optimizations to improve large file throughput in web server and remote-storage scenarios.

Generally, there are two solutions to remove the performance bottleneck caused by heavy memcpy operations. One solution is inspired by the zero-copy concepts, which is to eliminate the number of copy operations with software modifications. This solution removes the performance bottleneck completely but introduces other performance and security concerns, as illustrated in II-B. The other solution is to accelerate memcpy operations with the assistance of hardware. Hardware accelerators are a promising solution for relieving CPU pressure and dealing with specific workloads, and they usually lead to a great performance improvement. Some studies [11]–[14] have made great efforts to offload copy operations to hardware accelerators, such as GPU, IOAT, cache-based and processor-DMA-based accelerators. These studies mainly focused on how to utilize the characteristics of hardware accelerators to achieve a high speedup ratio without paying much attention to the performance issues caused by integrating hardware with real applications or systems. However, our study integrates hardware accelerators with an open-source network stack and finds that a straight integration method that simply offloads all copy requests to hardware and waits for completion cannot attain the anticipated performance. During the integration, we found there are two main performance issues caused by the straight solution. First, the synchronous architecture of this straight method leads to long CPU waiting time and is a severe waste of CPU resources. Second, some small copy requests may not be suitable for hardware acceleration for its high offloading overhead and little speedup. In addition, high hardware generality and availability are also demanded for a

§Corresponding Author

network system [15], [16].

To resolve these performance issues, in this paper, we re-engineer the copy architecture in the network stack and propose HAVS (hardware-accelerated VPP stack). To reduce the long CPU waiting time, HAVS introduces an asynchronous copy architecture to spend more CPU resources on protocol processing instead of waiting. Moreover, a decision-maker engine module is introduced in the architecture, which serves as a selector of copy methods and just offloads large copy requests to the hardware accelerator. Apart from addressing the high-performance requirement, HAVS also meets the requirements of high hardware generality and availability. Regarding high hardware generality, HAVS introduces an abstract memcpy accelerator layer, which provides uniform offloading interfaces to the upper network stack. Through implementing these interfaces, different ASIC-based hardware accelerators can be mounted to HAVS without awareness of the upper copy logic. Regarding the high availability requirement, HAVS implements a complete fault-tolerance mechanism that can detect faults of accelerators in time and switch the offloading path from a faulty accelerator to a normal accelerator during runtime.

In the detailed implementation, HAVS integrates Intel I/O Acceleration Technology into the VPP network stack to achieve high-performance memcpy offloading. We have implemented HAVS based on VPP and IOAT. VPP [5] is an open-source shared-memory-based userspace network stack proposed by Cisco and has become one of the mainstream userspace stacks in recent years. As a modern ASIC-based memcpy acceleration solution, Intel I/O Acceleration Technology introduced an asynchronous direct memory access (DMA) copy engine within the chip that has direct access to the main memory without high DMA startup and completion overheads.

In the evaluation, we deploy HAVS as a network stack to accelerate the web server application nginx and remote-storage application SPDK iSCSI target. Wrk, apachebench and fio tools are used to generate HTTP or IO requests to measure throughput and latency. In the web server experiment, HAVS reaches a 60% improvement over the original VPP network stack when the requested file size is larger than 4 KB with a 0.3% higher latency. Regarding small file cases, HAVS reaches a similar throughput, just 1-2% lower than the original VPP. In the SPDK iSCSI target experiment, HAVS reaches an average 53.5% throughput improvement in the random read case and reduces the average read latency by 20%-45%. In both scenarios, HAVS reaches a higher throughput per core than the kernel, 41.2% in the web server scenario and 45.9% in the remote-storage scenario.

In summary, this work makes the following contributions:

1) A verification experiment is performed to prove that the performance bottleneck of the shared-memory-based network stack exists in the copy process across shared memory.
2) An asynchronous copy architecture with a decision-maker engine is introduced to fulfil the high performance requirement.

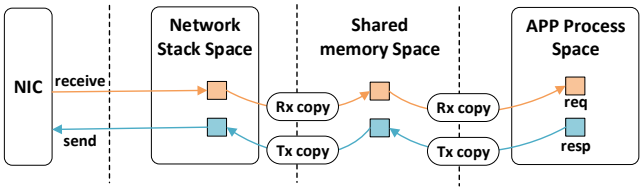

Fig. 1. Shared-memory-based Network Stack

3) An abstract memcpy accelerator layer with a complete fault tolerance mechanism is introduced to fulfil the high generality and availability requirements.
4) We show that HAVS can be practically used to accelerate the web server and remote-storage applications and experimentally evaluate its performance.

II. Background & Motivation

A. Performance Bottleneck Existing in the Memcpy Process

The number of requests to transfer large files via network has increased rapidly [17]. First, with a variety of multimedia files appearing on various webpages, the current average page size of over a million top sites has reached approximately 1400 KB [2]. Second, cloud providers separate computation and storage from inside a single physical machine to different machines for enhanced security and low TCO (total cost of ownership) [18]. This separation architecture necessitates that IO requests are transferred via the network. According to the report of the Live Optics program, the average IO transfer size between servers is 34.4K [19]. These large file transfer scenarios in web servers and remote-storage services pose a great challenge to the performance of the network stack.

Userspace network stacks have increasingly attracted attention [4]–[6], [20], [21] due to many inefficient problems in kernel network stack [22], [23]. When these userspace network stacks provide services to upper applications, a prevalent solution is to start the network stack as a separate host service, generally as an ordinary Linux process or a container, and communicate with applications with shared-memory queues. Currently, many mainstream userspace network stacks, such as VPP [5] and snap [6], adopt this solution for weak coupling with applications, full resource control of stacks and enhanced security brought by separating network stacks' memory from applications' memory.

However, this shared-memory solution leads to multiple memcpy operations between network stacks and applications. A normal workflow of the shared-memory network stack is depicted in Fig. 1. In the RX workflow, (1) the network stack receives the request packet from NIC. (2) After performing the necessary protocol processing, the stack enqueues the packet to a data queue in the shared memory. (3) The application dequeues the packet from the data queue and does some internal processing. The TX workflow is similar to the RX workflow and also needs two copy operations for enqueuing and dequeuing packets returned. Intuitively, these extra copy

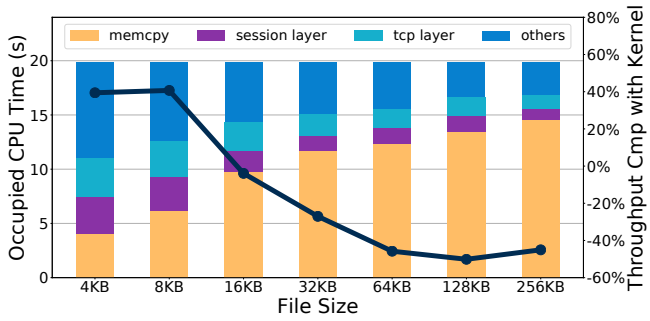

Fig. 2. CPU Utilization and Performance Gap with Kernel

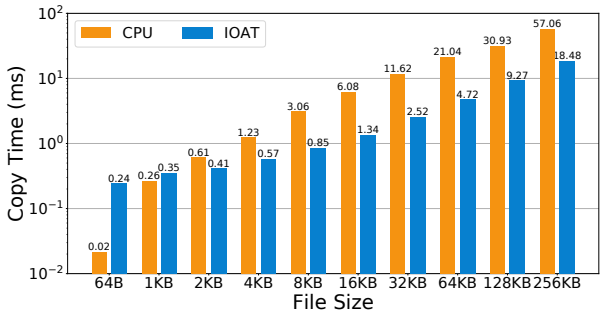

Fig. 3. Average Copy Time of CPU and IOAT Hardware

operations for communication will become the main performance bottleneck, especially when transferring large files.

To verify the performance bottleneck brought by memcpy operations, we evaluated the CPU utilization of an open source network stack (i.e., VPP) by the vtune [24] tool on two servers connected via four Intel XL710 40GbE NICs. First, we start a VPP stack with one worker core providing services to the upper nginx application with four worker cores on one server and send HTTP requests from the other server by the wrk tool. Second, we measure the CPU execution time of all functions in the VPP stack by running the vtune profiler for 20 seconds during HTTP request processing. Finally, we change the data size sent by the wrk tool and repeat the above experiment to find the relations between the CPU execution time and requested data size. In the results shown in Fig. 2, it is observed that memcpy operations always take up the most percentage of CPU time compared with the other two high time-consuming tasks, TCP protocol processing and session layer processing. Meanwhile, CPU resources occupied by memcpy operations increases greatly as the requested file size becomes large. When the requested file size is less than 16-KB, memcpy operations consume approximately 20%-25% of CPU resources. However, when the requested file size exceeds 64 KB, memcpy operations consume more than 60% of CPU resources. To further verify that the high CPU utilization of memcpy operations leads to severe performance degradation of VPP, we start five nginx workers with the kernel network stack as a contrast and measure the throughput of the VPP approach and kernel approach by the wrk tool. As shown in the curve in Fig. 2, VPP approach achieves an approximate 40% improvement in 4 KB and 8 KB cases while a 40% degradation in 64 KB, 128 KB and 256 KB cases.

B. Zero-Copy or Hardware-Assisted Copy

To eliminate the CPU utilization of memcpy operations in the shared-memory network stack, generally, there are two possible solutions inspired by previous work [12], [25]. One solution is called zero-copy, which is a general method used in the case of DPDK [26] to remove memcpy operations in the system. The other solution, which is always implemented with the assistance of other hardware, is to speed up memcpy operations instead of removing them. These two solutions eliminate the performance bottleneck caused by memcpy but have their own limitations. Regarding the zero-copy method, it is definitely an ideal solution but leads to security and other performance concerns. Once zero-copy is enabled, everything will be mapped into the VPP address space. In this situation, malicious applications could pollute the internal rather than temporary memory of the network stack with a buffer overflow attack. In this way, it could also pollute the memory of other applications supported by VPP stack. Moreover, as mentioned in snap [6], zero-copy needs frequent memory management operations during runtime, which may lead to other performance concerns such as TLB shootdown. Regarding the hardware-assisted method, it retains the original structure of the shared-memory network stack and adds few extra security problems or performance concerns. However, this method can be used only in the physical machine with specific hardware for memcpy acceleration. In our work, HAVS adopts the second solution to use IOAT as the hardware accelerator and leaves zero-copy as future work.

C. Intel I/O Acceleration Technology

Intel I/O Acceleration Technology (IOAT) [27] is an ASIC-based solution to enable data copy by the chipset instead of the CPU. In this way, IOAT not only moves data more efficiently but also frees up CPU resources in the system.

1) Related Work with IOAT Integration: Originally, IOAT was designed for the kernel network stack and used in the Windows Server 2003 OS. However, in recent years, it has been integrated into DPDK and can be accessed directly through the userspace driver, such as VFIO and UIO drivers. Due to this convenient userspace access model, IOAT has been integrated into the para-virtual framework [28], SPDK [29] and snap [6] to accelerate data transferring. Snap is also a shared-memory-based network system that implements a range of forward-looking network functions. IOAT is integrated in snap for memcpy acceleration and achieves an approximate 22.5% performance boost in the system. However, snap is not an open-source system and is limited to support Google's internal application, such as grpc [30]. Our work is inspired by snap and does a further analysis to the memcpy bottleneck in shared-memory-based network stack. We adopt an open-source VPP network stack and support the prevalent applications,

nginx and SPDK iSCSI target. In contrast to the previous IOAT use in SPDK, we accelerate the network stack instead of the internal storage system with IOAT. After a series of optimizations, we achieve an approximate 50%-60% performance improvement compared with the original VPP stack.

2) Performance Comparison of Memcpy and the CPU: Before this work, we compare the speed of the CPU memcpy with memcpy accelerated by IOAT. In the experiment, we put five hundred memcpy requests of the same data size into IOAT and perform constant polling until copy requests are all completed. As a contrast, CPU memcpy operations will also be executed five hundred times. In the end, the average time for a copy request is calculated using total time divided by five hundred. To minimize the influences caused by cache, we make all requests copy from different source addresses to different destination addresses. In addition, we change the copy size in the experiment. The final evaluation is depicted in Fig. 3. IOAT performs better when the copy size is larger than 1 KB while the CPU performs better when the copy size is smaller. Regarding the large sizes, such as 64 KB, 128 KB and 256 KB, IOAT could improve the performance of memcpy by 4.46x, 3.34x, 3.09x, respectively.

III. Challenges with Memcpy Offloading

HAVS offloads memcpy operations in VPP, an open source shared-memory network stack, to IOAT hardware. In our work, there are two aspects of challenges existing in this offloading process, which are the high-performance requirements of VPP stack and the easy-to-use offloading abstraction with guarantee of high hardware generality and availability.

A. High-Performance Requirement

1) How to Free Up CPU Resources in VPP Stack: A straight offload approach for memory copy is to replace the CPU memcpy function call with an I/O call that interacts with IOAT hardware. This approach reuses the existing VPP stack with only a few modifications. However, this kind of synchronous copy leads to a severe waste of the CPU because the saved CPU resources in charge of copying are simply spent on waiting for IOAT copy responses. This under-utilization of computation resources cannot reach the anticipated performance enhancement. Therefore, a well-designed asynchronous copy architecture is needed.

2) Performance Concern about Small Packets: As depicted in Fig. 3, IOAT hardware takes more time to complete the copy process for a small size of data even if it has a great advantage over the CPU in the case of a large size. Although the packet size could be enlarged by enabling tso to break the limit of Ethernet MTU (1500 Byte), a straight method to simply offload all copy requests to IOAT is not suitable in the small-packet-dominated scenario. Therefore, it appears that returning small copy requests to the CPU and only offloading large copy requests to hardware is a smarter choice. In addition, the CPU copy is a synchronous process while IOAT copy is an asynchronous process. It may happen that a copy request completed by the CPU has to wait a long time for the requests previously offloaded to IOAT to complete for the order-preserving property. Thus, a lightweight decision-maker engine to dispatch copy requests and minimize latency for CPU-copied packets is needed.

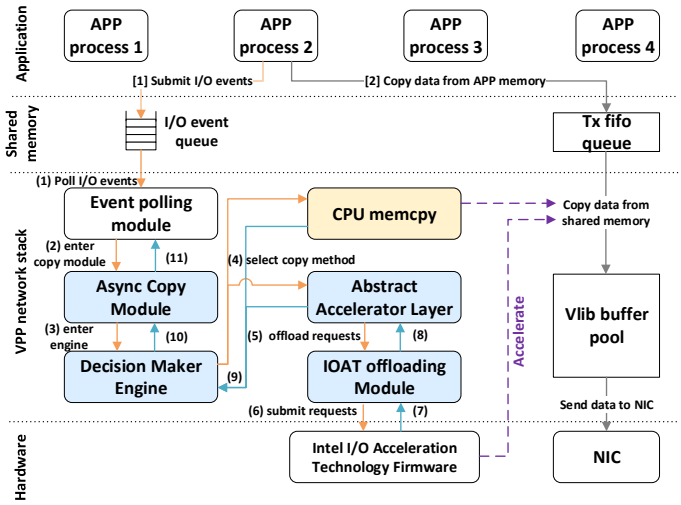

Fig. 4. HAVS Architecture

B. High Generality and Availability Requirement

1) Support More Hardware Accelerators: Although IOAT is selected as the hardware accelerator in our work, there are still other ASIC-based hardware accelerators in different chipsets. In addition, some hardware for memcpy acceleration is in the development phase and may be published in the near future. These other hardware accelerators could accelerate VPP in the servers without IOAT or in the future, but the whole software stack needs to be re-developed without a well-designed accelerator abstraction. To easily support more hardware accelerators, an abstraction for hardware accelerators needs to be proposed and a uniform interface for most accelerators needs to be provided.

2) High Availability of IOAT Hardware: Similar to many hardware devices, IOAT hardware may encounter faults for some reasons, such as the number of copy requests exceeding the hardware load. To retain high availability of the VPP stack accelerated by IOAT, a fault-tolerance mechanism is essential to find an alternative memcpy method when IOAT hardware is in the fault state.

IV. System Design

In this paper, based on the VPP network stack and Intel I/O acceleration technology, we propose HAVS, which offloads memcpy operations for TCP packets to IOAT hardware. The overall IOAT acceleration subsystem in HAVS, including Async Copy Module, Decision-Maker Engine (DME), Abstract Accelerator Layer (AMA Layer) and IOAT offloading Module, is depicted in Fig. 4. HAVS mainly accelerates the copy process from shared memory to the private memory of VPP. The modification starts with the event polling module in

the VPP session layer that polls I/O requests from applications. Instead of the software memcpy path, HAVS forwards I/O requests to the IOAT acceleration subsystem. In the subsystem, the async copy module is responsible for some preprocessing, such as address translation. DME is in charge of dispatching copy requests. Moreover, HAVS introduces the AMA layer to address the hardware generality and availability requirement. The IOAT offloading module encapsulates the API of the IOAT library in DPDK.

To explain the functions of these HAVS modules, the memory copy workflow is depicted in Fig. 4. (1)-(2): The session node forwards I/O requests to the async copy module. (3): the async copy module performs some preprocessing and forwards requests to DME for selection of copy methods. (4): If the hardware-assisted copy method is selected, it will forward requests to the AMA layer, which initializes and registers hardware accelerators in advance. (5): The AMA layer will then submit the requests to the corresponding hardware offloading module. (6)-(7): The offloading module enqueues copy requests to hardware rings and consumes responses to check if requests enqueued are completed. (8)-(11): The completion results are returned and go through the upper modules one by one. In the end, the async copy module receives the responses, does some postprocessing and forwards copy-completed packets to the next VPP processing node.

The combination of the Async Copy Module and DME provide an asynchronized and hybrid copy architecture. The asynchronized copy architecture allows the CPU to handle other tasks by offloading the memcpy task to hardware. During the hardware copy process, the CPU continues to poll new I/O events from applications and executes protocol-related logic, such as adding headers for packets. Moreover, to fix the problem of low speed when IOAT copies small packets, HAVS introduces DME to serve as a selector of copy methods. Taking the copy time of the CPU and IOAT prefetched in Fig. 3 as a reference, DME dispatches different copy requests to the CPU or hardware accelerator. In addition, a temporary buffer structure called the inflight buffer for storing uncompleted data is designed to reduce the latency of CPU-copied packets.

The main function of the AMA layer is to decouple upper copy logic from the underlying hardware offloading module for high hardware generality and high availability. The AMA layer defines a suite of uniform hardware-accelerated-copy interfaces, by which upper logic first offloads copy requests to virtual accelerators. Then, the virtual accelerator submits copy requests to the real hardware accelerators. Based on the AMA layer, different hardware accelerators can be easily mounted to HAVS by simply implementing interfaces using their own offloading API. On the other hand, the AMA layer introduces a complete fault-tolerance mechanism during the process in which virtual accelerators submit requests to real hardware. When different faults, including the number of requests exceeding the limit of hardware, requests with illegal src/dst addresses being submitted and other hardware faults occur, HAVS could retain high availability by returning requests back to the CPU or replacing the faulty accelerator with a normal one immediately.

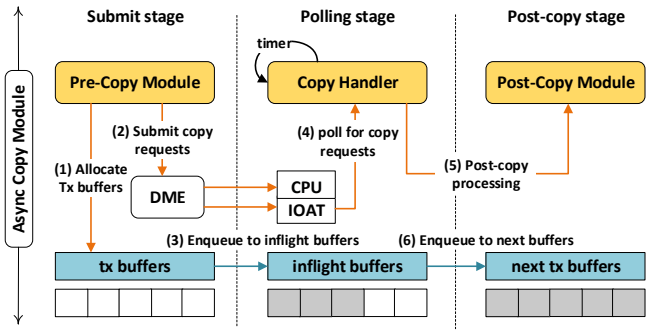

Fig. 5. Asynchronous Copy Architecture

V. Implementation & Optimization

The implementation of the HAVS prototype relies on the VPP network stack and IOAT hardware for memcpy acceleration. In this section, we present the implementation of HAVS in detail. Section V-A mainly introduces the upper logic implemented in the VPP stack regarding performance optimizations, while Section V-B mainly focuses on the offloading implementation with high hardware generality and availability.

A. Accelerating Memcpy in the VPP Stack

1) Asynchronous Copy Workflow: The VPP stack is composed of many processing nodes. Each node implements a part of network protocols and the whole stack is constructed by connecting these nodes to a node graph. Among these nodes, the session queue node is mainly in charge of the copy process from shared memory to the VPP stack. To free up CPU resources, HAVS implements an asynchronized copy architecture in the session queue node. The detailed asynchronized workflow, depicted in Fig. 5, is divided into three stages, submit stage, polling stage and post-copy stage. The submit stage mainly offloads copy requests to IOAT hardware. First, HAVS will allocate a sufficient number of tx buffers for storing copied packets. Then virt-to-phys address translation is executed and HAVS packs copy parameters into a request. Next, these requests are submitted to DME, and DME will select a further copy method for each request. In the polling stage, HAVS polls the completed requests from IOAT hardware when inflight requests exceed a specific threshold, set to 10 by default. A timer is also set to periodically check whether at least one polling operation is triggered during the last interval, and if this has not occurred but there are still inflight requests, an extra polling operation can be executed at once. Next, when completed requests are polled from IOAT, HAVS enters the post-copy stage and does some protocol-related processing including setting the retransmission timer. Finally, HAVS enqueues the completed buffers to the next tx buffers. It means the next processing node in VPP will take over the following processing.

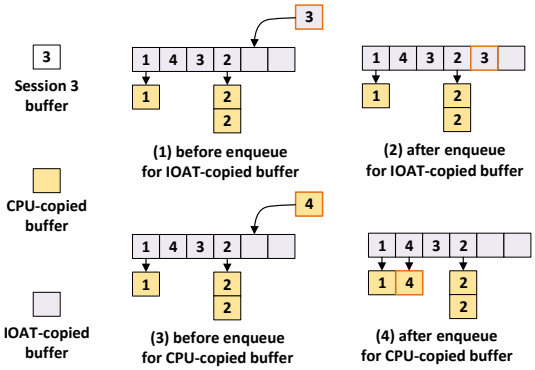

Fig. 6. Inflight Buffer Design

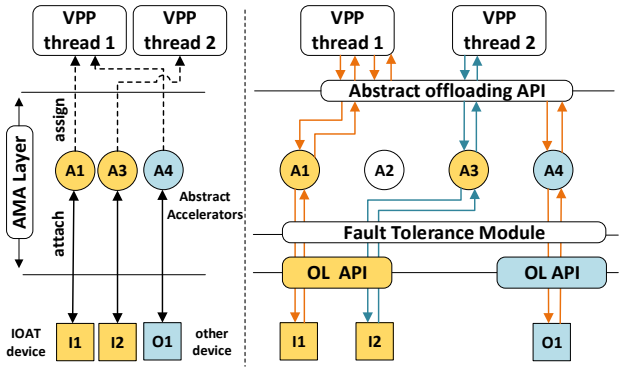

Fig. 7. Abstract Memcpy Accelerator Layer

2) Copy Decision Maker: According to Fig. 3 and further experiment, a fact is observed that CPU performs much better at copying small packets than IOAT hardware. To fully utilize the advantage of IOAT when copying large data, we support tso and enlarge the size of the buffer structure used to store packets in VPP. Regarding the the small requested packets that cannot be enlarged by tso, HAVS provides a hybrid copy approach and offloads different copy requests to the CPU or IOAT according to the size of data. In HAVS, all copy requests need to pass by DME for dispatching. DME determines the copy strategy for each request according to the copied data length. If the length exceeds a specific threshold, DME will offload the request to IOAT hardware. Otherwise, it will adopt a synchronized copy method by the CPU. HAVS adds the threshold as an option to the VPP configuration file. Users can customize this threshold to balance the load of the CPU and IOAT. By default, HAVS sets the threshold to 1 KB for the high-performance requirement. Because the DME is very lightweight with just a few determine statements, it will not introduce a high performance penalty in the system. In addition, DME also supports customizing determine statements according to characteristics of other hardware accelerators.

Latency-aware Inflight Buffers: Although the copy process can be completed by the IOAT hardware or CPU, their buffer indexes both need to be enqueued into the inflight buffer due to the order-preserving property required by TCP protocol. However, CPU copy is a synchronized process while IOAT hardware copy is an asynchronized process. It means buffers completed by CPU copy may have to wait for buffers previously offloaded to IOAT hardware to be completed. HAVS would like to minimize the waiting time for CPU-copied buffers. The inflight buffer is designed to send CPU-copied buffers to the next buffers for further processing as early as possible. It implements two different enqueue modes for two copy methods, as depicted in Fig. 6. (1)-(2) is the enqueue mode for IOAT hardware. It just adds the buffer into the end of the queue of the inflight buffer because IOAT copy is sequential and the buffers offloaded previously must be completed first. (3)-(4) is the enqueue mode for CPU-copied buffers. Because they do not have to wait for buffers of other sessions and their copy process has already finished, HAVS enqueues the buffer to the location just after the latest buffer of the same session instead of the end of the queue. In this way, buffers copied by the CPU can be sent out immediately once the latest buffer of the same session copied by IOAT hardware is completed and there is no need to wait for buffers of other sessions.

B. Offloading to Accelerators

To ease use of hardware accelerators, HAVS introduces an AMA (Abstract Memcpy Accelerator) layer, which contains a pool of virtual memcpy accelerators. As depicted in Fig. 7, each hardware accelerator is initialized and attached to one virtual accelerator when HAVS boots up. Then, HAVS assigns these virtual accelerators to different VPP threads. In HAVS, the virtual accelerator serves as the agent of the hardware accelerator. During runtime, the virtual memcpy accelerator receives copy requests from DME through a uniform offloading interface and forwards them to the real hardware by a hardware-specific offloading API in the downstream workflow. The upstream workflow is similar to the downstream workflow. The virtual accelerator obtains the status of hardware and returns it to the upper polling handler. A complete fault-tolerance mechanism is introduced in the layer to retain high availability for HAVS.

1) Uniform Interface for Offloading: The memcpy accelerator layer provides a suite of uniform offloading interfaces to the upper network stack for submitting requests and consuming responses. Different types of hardware could be integrated into HAVS through implementing these interfaces with no awareness of the upper network stack. Considering possible differences between different types of hardware, HAVS simplifies the interfaces and only lists the most important part of memcpy offloading. There are four main offloading interfaces: (1) configure: Pass the configuration to accelerator in the initialization stage. (2) enqueue_copy: Enqueue the copy requests to accelerator's descriptor ring. (3) dequeue_copy: Dequeue the completed copy requests from the accelerator's

descriptor ring. (4) peek_copy: Peek_copy is similar to the dequeue_copy interface but it will not free the completed requests in the descriptor ring.

2) Fault-Tolerance Mechanism: Generally, there are two cases for unavailability of IOAT. One case is called temporary unavailability. It always happens when the number of inflight copy requests exceeds the limit of the hardware accelerator under a high load. In this case, the accelerator could not accept more requests until some of the inflight requests are freed. Once it happens, instead of waiting for the accelerator to complete some inflight requests, HAVS submits this copy request to the CPU immediately. This approach could help to relieve the request pressure of IOAT. Meanwhile, HAVS records the times of this temporary unavailable case during a period. If the number of times exceeds a specific threshold in a short time, another accelerator is assigned to the overwhelmed VPP thread.

The other case is called permanent unavailability, which is mainly caused by a hardware fault or illegal copy requests, such as copying a segment of physically discontinuous memory. In this case, one hardware accelerator without a self-reboot function retains the unavailable state until the server is rebooted. IOAT is one type of these accelerators. If there are no other hardware accelerators in the system, HAVS will go back to the CPU copy process. Otherwise, HAVS will detach the virtual accelerator from the faulty hardware and re-attach normal hardware to it. In the meantime, inflight copy requests uncompleted by the faulty hardware will also be re-submitted to the new hardware. Then, HAVS returns to the normal process of copying. During this runtime switch process, the upper VPP stack is unaware of the hardware change in the AMA layer. If new hardware also breaks down in this switch process, HAVS can identify that this fault is caused by illegal copy requests and report an error in the system before stopping all VPP services.

VI. Evaluation

A. Experimental Setup

We established an experimental testbed with one tested server and one client server that were connected back-to-back via two pairs of Intel XL710 40GbE NICs. The tested server was equipped with two 28-core Intel XZeon Platinum 8280L CPUs (hyperthreading disabled) while the client server was equipped with two 18-core Intel Xeon Gold 6140M CPUs. In addition, they were both equipped with 64 GB RAM. We ran Ubuntu 18.04 with Linux Kernel 4.15 on them. HAVS was installed on the tested server with eight IOAT devices per numa node.

The HAVS testbed was based on VPP in two different scenarios. One is to provide services to the web server. In this scenario, we configure nginx to serve as a HTTP server based on VSAP project [31]. The other is to provide services to storage applications. In this scenario, we select the SPDK iSCSI target application to set up a remote storage environment. In both scenarios above, VPP is configured to provide TCP services for remote wget or I/O operations.

In the tested server, nginx workers, SPDK threads and VPP workers were each configured to occupy a dedicated physical core. Each VPP worker started as a thread and was equipped with one or more virtual accelerators. The allocated virtual accelerator was attached to one IOAT device. In addition, we bound the VPP worker with the IOAT device in the same numa node in the experiment.

The evaluation was conducted in terms of data transfer throughput, core efficiency and average response time. To explain the latency results, we performed extra two experiments for deep analysis. Moreover, small-file request cases are designed to prove the advantages of DME, and overhead brought by the fault-tolerance mechanism is measured accurately. To evaluate all aspects of HAVS, the following four configurations are compared.

- VPP SW: copy data from shared memory by AVX2 memcpy operations.
- IOAT+S: straight offload approach using synchronous copy process.
- IOAT+A: asynchronous offload framework with timer-based polling.
- HAVS: the full HAVS with asynchronous offload approach with both timer-based polling and the decision-maker engine.

B. Web Server Application

In the scenario of the web server, we used wrk [32], a modern HTTP benchmarking tool, to measure the data transfer throughput with the requested file size varying from 0 KB to 256 KB. In the tested server, four nginx workers were launched and the keep alive setting was enabled to avoid the influence of the connection setup and teardown. In the client server, 36 wrk processes were configured and 1800 connections were set up to continuously request for a fixed file.

1) Data Transfer Throughput: The experimental results from the 1 KB file size to the 256 KB file size are shown in Fig. 8. Throughput value increases linearly for all four configurations when the requested file size varies from 1 KB to 256 KB. Taking the 64 KB case as an example, the IOAT+S configuration provides 22.45 Gb of throughput. In the IOAT+A configuration, the asynchronous offload framework fully frees up CPU resources spent on waiting and brings a throughput boost of 60.57 Gb, which is a 2.7x improvement over IOAT+S and a 61% improvement over the VPP SW configuration. In the HAVS, the overhead incurred by the request dispatching logic of the decision-maker engine does not lead to much of a performance drop. Generally, the full HAVS provides an average of 60% throughput improvement over the software baseline when the requested file size is larger than 4 KB.

2) Core Efficiency: The application and network stack occupy the same cores when running the application with the kernel network stack, so it is hard to compute how many CPU resources are occupied by the kernel stack alone. Moreover, the kernel network stack always schedules some tasks to other cores under a high workload during runtime. To compare the HAVS performance with the kernel fairly, we define a new

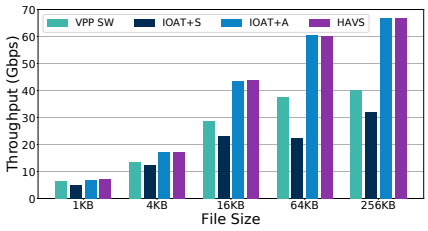

Fig. 8. Throughput (NGINX)

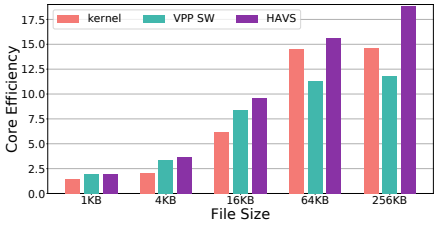

Fig. 9. Core Efficiency (NGINX)

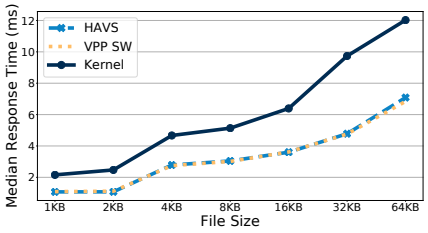

Fig. 10. Median Response Time (NGINX)

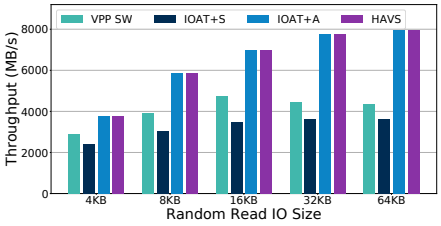

Fig. 11. Throughput (SPDK ISCSItgt)

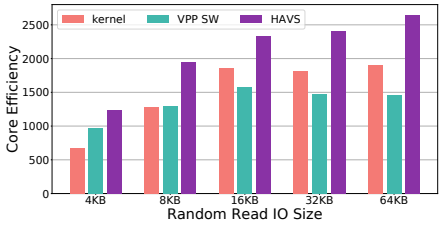

Fig. 12. Core Efficiency (SPDK ISCSItgt)

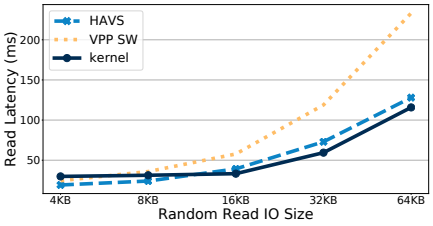

Fig. 13. Read Latency (SPDK ISCSItgt)

comprehensive metric called core efficiency, which is equal to the throughput divided by the sum of the core utilization occupied by the application and network stack.

In the experiment, four nginx workers and one VPP worker are started when tested with VPP and HAVS, while five nginx workers are started when tested with the kernel. The wrk tool is used to measure requests per second with different requested file sizes. The runtime CPU utilization is measured by htop, the most popular monitoring tool in the Linux system. As depicted in Fig. 10, the core efficiency increases linearly with the requested file size varying from 1 KB to 256 KB. This is because larger requests often lead to higher throughput with fewer extra overheads. HAVS reaches the highest core efficiency in all cases, reaching a 25.2% improvement over VPP SW and a 41.2% improvement over the kernel on average. The gaps of the core efficiency between HAVS and VPP SW or the kernel widen when the requested file size becomes large.

3) Median Response Time: We used Apachebench (ab) to measure the median response time with different requested file sizes. In this experiment, only one nginx worker was launched in the tested server. In the client server, multiple ab processes were launched to make concurrent requests for files.

As shown in Fig. 10, the value of the average response time increases linearly when the requested file size increases. Compared with the kernel stack, HAVS reaches an average 51.0% lower response time when the requested file size varies from 0 KB to 64 KB. In addition, HAVS reaches a similar response time with VPP SW when tested with the same size of requests, approximately 0.3% higher on average. This is a compromised result caused by the faster speed of the IOAT hardware and extra overhead brought by the asynchronous architecture, which we will illustrate in VI-D.

C. Remote Storage Application

In this scenario, we used the FIO (Flexible I/O tester) [33] tool to measure bandwidth and latency with different block sizes. In the tested server, 32 block servers were launched in the SPDK iSCSI target application [34] and two cores are assigned to SPDK. In the client server, because the main optimization of HAVS focuses on the TX path of VPP, we configure the FIO tool to perform random read operations to 32 block servers concurrently. We will leave the RX path acceleration as future work.

1) Data Transfer Throughput: We start one VPP worker in the tested server and other configurations are similar to the web server experiment. The experimental results from 4 KB IO size to 64 KB IO size are shown in Fig. 11. From the Fig, the throughput of HAVS is similar to the IOAT+A configuration and reaches the limit of two NICs in the 64 KB case. To sum up, HAVS reaches a 53.5% performance improvement over VPP SW and a 93.5% improvement over IOAT+S on average.

2) Core Efficiency: In the experiment, three cores are assigned to SPDK when tested with the kernel, while one core is assigned to VPP and two cores are assigned to SPDK when tested with VPP SW and HAVS. As depicted in Fig. 12, VPP SW performs better than the kernel only in the 4-KB case. With the IO size becomes larger, HAVS achieves a larger performance boost compared with the other two configurations. HAVS reaches the highest core efficiency in all cases, reaching a 53.6% improvement over VPP SW and a 45.9% improvement over the kernel.

3) Random Read Latency: As shown in Fig. 13, the read latency increases with the IO size varying from 4 KB to 64 KB. In the case of 4 KB, the random read latency of VPP SW, kernel and HAVS is similar, but the kernel and

TABLE I
REQUESTS COMPLETED RATE (RCR) TABLE

FS (KB)	2	4	8	16	32	64	128
RCR (%)	97	93	89	91	82	55	41

TABLE II
TIME INTERVAL BETWEEN TWO POLLING OPERATIONS

FS (KB)	2	8	64	128
VPP SW (ms)	88.3	85.6	92.2	101.5
HAVS (ms)	86.2	77.0	71.6	64.9
Diff (%)	-2.34	-9.97	-22.34	-36.12

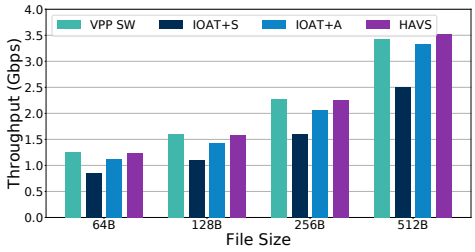

Fig. 14. Small File Throughput (NGINX)

HAVS perform much better in large-size cases. Due to the zero-message optimization in SPDK, the kernel achieves low latency when the IO size is larger. HAVS reaches a latency result that is comparable to the kernel due to the fast speed of the IOAT hardware. In the 64 KB case, the random read latency of HAVS is 10.5% higher than that of the kernel but 45.1% lower than that of VPP SW.

D. Further Analysis of Latency

The asynchronous copy architecture in HAVS intuitively leads to longer request latency, but the test result shows HAVS reaches similar or even lower latency compared with VPP SW in two scenarios. To further investigate this result, we take the web server scenario as an example and collect the completed rate of copy requests offloading to the IOAT when the first polling executes. As shown in Table I, we observe that more than 90% copy requests are completed before the first polling when the requested size is below 16 KB. Because the first polling happens just after the offloading process, it means more than 90% of copy requests are completed during the offloading process and sent out immediately as in the synchronous mode, which leads to low latency overhead. For the large file size, such as 128 KB, only 41% are completed during the first polling. However, because of the faster copy speed achieved by IOAT in large-size cases, the time interval between two pollings in HAVS is much shorter than in VPP SW. As shown in Table II, taking 128 KB as an example, the time of one loop in HAVS is 36.1% shorter than in VPP SW, which compromises the overhead caused by the asynchronous architecture.

E. Decision-Maker Engine

To prove the advantages of DME in HAVS, we test the throughput of small files in the web server scenario. The experimental results from 64-B to 512-B file sizes are shown in Fig. 14. The VPP SW configuration provides the highest throughput in most cases. Taking the 128-B case as an example, the VPP SW achieves 1.61 Gbps throughput while IOAT+S and IOAT+A achieve 1.10 and 1.42 Gbps throughput, which are 31.7% and 11.8% lower respectively. The slower copy speed of IOAT hardware in the small-size case and extra overhead caused by offloading operations together lead to this performance degradation. HAVS with DME does a smart dispatch and returns these small packets back to the CPU. In this way, HAVS could achieve a similar throughput performance with VPP SW. In the 128-B case, the throughput of HAVS is only approximately 2% lower than the VPP SW.

F. Overhead of Fault Tolerance

We evaluated the overhead of the fault-tolerance mechanism in HAVS to assure that it would not lead to much performance degradation. The overhead is mainly caused by recording every request enqueued into the IOAT hardware during runtime. The rdtsc system call instruction is used to obtain the current CPU cycles from the time register, tsc. In our experiment, we measure the CPU cycles used to execute request recording operations with the help of rdtsc. The results show that approximately 66 extra cycles are occupied per offloading process, which leads to an average 0.31% throughput degradation.

When the runtime hardware switch is triggered by the hardware fault, there exists a service unavailable period until the process is finished. We measure the downtime of VPP service using rdtsc and runtime CPU frequency. The results shows the switch process consumes 5.7 us on average.

VII. CONCLUSION

In this paper, we proposed and implemented HAVS, an asynchronous memcpy acceleration framework with high performance, generality and availability based on the IOAT hardware accelerator and VPP network stack. HAVS targets the problem that memcpy has become the main performance bottleneck of the shared-memory-based network stack and solves it by two layers of hardware offloading. At the upper layer, HAVS introduces an asynchronous copy architecture with a smart decision-maker engine to maximize the performance of the CPU and IOAT hardware. At the bottom layer, HAVS introduces an abstract memcpy accelerator layer to support more hardware accelerators in the system and reaches high availability with a complete fault-tolerance mechanism. In the evaluation, HAVS was used to accelerate the nginx and SPDK iSCSI target application and outperformed the VPP network stack by 50%-60% with a similar or even reduced latency.

ACKNOWLEDGEMENT

This work is supported in part by the National Natural Science Foundation of China (No. 61972245).

REFERENCES

[1] Y. Yubing and L. Zhanping, "Research of fast and safe large files transmission based on socket," *ICCSE*, pp. 483–485, 2012.

[2] "Webpages are getting larger every year." [Online]. Available: https://www.pingdom.com/blog/webpages-are-getting-larger-every-year-and-heres-why-it-matters/#, accessed July 2020

[3] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: a highly scalable user-level tcp stack for multicore systems," in *NSDI*, 2014.

[4] "F-stack: High-performance network framework based on dpdk." [Online]. Available: http://www.f-stack.org/

[5] "Introduction to vector packet processing (vpp)." [Online]. Available: https://fd.io/docs/vpp/master/

[6] M. Marty, M. Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat, "Snap: a microkernel approach to host networking," *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[7] "Nginx." [Online]. Available: https://www.nginx.com/

[8] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "Ix: A protected dataplane operating system for high throughput and low latency," in *OSDI*, 2014.

[9] G. Prekas, M. Kogias, and E. Bugnion, "Zygos: Achieving low tail latency for microsecond-scale networked tasks," *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[10] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive scheduling for μsecond-scale tail latency," in *NSDI*, 2019.

[11] J. Lee, Z. Liu, X. Tian, D. H. Woo, W. Shi, and D. Boumber, "Acceleration of bulk memory operations in a heterogeneous multicore architecture," *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 423–424, 2012.

[12] K. Vaidyanathan, L. Chai, W. Huang, and D. K. Panda, "Efficient asynchronous memory copy operations on multi-core systems and i/oat," *2007 IEEE International Conference on Cluster Computing*, pp. 159–168, 2007.

[13] F. Duarte and S. Wong, "Cache-based memory copy hardware accelerator for multicore systems," *IEEE Transactions on Computers*, vol. 59, pp. 1494–1507, 2010.

[14] W. Su, L. Wang, M. Su, and S. Liu, "A processor-dma-based memory copy hardware accelerator," *2011 IEEE Sixth International Conference on Networking, Architecture, and Storage*, pp. 225–229, 2011.

[15] W. Wang and J. Doucette, "Availability optimization in shared-backup path protected networks," *IEEE/OSA Journal of Optical Communications and Networking*, vol. 10, pp. 451–460, 2018.

[16] C. She, Z. Chen, C. Yang, T. Q. S. Quek, Y. Li, and B. Vucetic, "Improving network availability of ultra-reliable and low-latency communications with multi-connectivity," *IEEE Transactions on Communications*, vol. 66, pp. 5482–5496, 2018.

[17] M. Aihara, S. Kono, K. Kinoshita, N. Yamai, and T. Watanabe, "Joint bandwidth scheduling and routing method for large file transfer with time constraint," *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pp. 1125–1130, 2016.

[18] Y. Zhang and Y. Zhou, "Separating computation and storage with storage virtualization," *Computer Communications*, vol. 34, pp. 1539–1548, 2011.

[19] "average io transfer size over of millions of servers." [Online]. Available: https://support.liveoptics.com/hc/en-us/articles/229590747-How-IO-Transfer-Size-Effects-latency-and-IOPS, accessed July 2020

[20] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo, "Rekindling network protocol innovation with user-level stacks," *Computer Communication Review*, vol. 44, pp. 52–58, 2014.

[21] "Seastar: open-source c++ framework for high-performance server applications." [Online]. Available: http://seastar.io

[22] S. Han, S. Marshall, B. Chun, and S. Ratnasamy, "Megapipe: A new programming interface for scalable network i/o," in *OSDI*, 2012.

[23] L. Soares and M. Stumm, "Flexsc: Flexible system call scheduling with exception-less system calls," in *OSDI*, 2010.

[24] "Intel vtune profiler user guide." [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top.html, accessed July 2020

[25] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa, "Pin-down cache: a virtual memory management technique for zero-copy communication," *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pp. 308–314, 1998.

[26] Intel, "Data plane development kit," 2014. [Online]. Available: https://www.dpdk.org/

[27] "Accelerating high-speed networking with intel i/oat." [Online]. Available: https://www.intel.com/content/www/us/en/io/i-o-acceleration-technology-paper.html?wapkw=I%2FOAT, accessed July 2020

[28] "Accelerating para-virtual i/o with cbdma (ioat hardware)," 2018, https://www.dpdk.org/wp-content/uploads/sites/35/2018/12/JiayuHu_Accelerating_paravirtio_with_CBDMA.pdf, accessed July 2020.

[29] "Fast memcpy with spdk and intel i/oat dma engine." [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/fast-memcpy-using-spdk-and-ioat-dma-engine.html

[30] "grpc." [Online]. Available: https://www.grpc.io/

[31] Intel, "Vsap: Vpp stack acceleration project." [Online]. Available: https://wiki.fd.io/view/VSAP

[32] "wrk tool." [Online]. Available: https://github.com/wg/wrk

[33] J. Axboe, "Welcome to fio's documentation, 2017." [Online]. Available: https://fio.readthedocs.io/en/latest

[34] "Spdk iscsi target getting started guide." [Online]. Available: https://spdk.io/doc/iscsi.html