FastFCoE : An Efficient and Scale-up Multi-core Framework for FCoE-based SAN Storage Systems

Yunxiang Wu, Fang Wang*, Yu Hua, Dan Feng, Yuchong Hu, Jingning Liu, Wei Tong
Wuhan National Laboratory for Optoelectronics
School of Computer, Huazhong University of Science and Technology, Wuhan, China
{yxwu, wangfang, csyhua, dfeng, yuchonghu, jnliu, tongwei}@hust.edu.cn

***Abstract*—Due to the high complexity in software hierarchy and the shared queue & lock mechanism for synchronized access, existing I/O stack for remote target access in FCoE-based SAN storage becomes a performance bottleneck, thus leading to a high I/O overhead and limited I/O scalability in multi-core servers. For scalable performance, existing works focus on improving the efficiency of lock algorithm or reducing the number of synchronization points to decrease the synchronization overhead. However, the synchronization problem still exists and leads to a limited I/O scalability.**

In this paper, we propose FastFCoE, a protocol stack framework for remote storage access in FCoE based SAN storage. FastFCoE uses private per-CPU structures and disables the kernel preemption to process I/Os. This method avoids the synchronization overhead. For further I/O efficiency, FastFCoE directly maps the requests from the block-layer to the FCoE frames. A salient feature of FastFCoE is using the standard interfaces, thus supporting all upper softwares (such as existing file systems and applications) and offering flexible use in existing infrastructure (e.g., adaptors, switches, storage devices). Our results demonstrate that FastFCoE achieves efficient and scalable I/O throughput, obtaining 1107.3K/831.3K IOPS (5.43/4.88 times as much as Open-FCoE stack) for read/write requests.

***Keywords*—Fiber Channel over Ethernet; Multi-core framework; Storage architecture;**

I. Introduction

In order to increase multi-core hardware utilization and reduce the total cost of ownership (TCO), many consolidation schemes have been widely used, for example server consolidation through virtual machine technologies and I/O consolidation through converged network adapters (CNAs, which combine the functionality of a host bus adapter (HBA) with a network interface controller (NIC)). The Fiber Channel over Ethernet (FCoE) standard [1], [2], [3] allows the Fibre Channel storage area network (SAN) traffic to be consolidated in a converged Ethernet without additional requirements for FC switches or FCoE switches in data centers. Currently converged Ethernet has the advantages of availability, cost-efficiency and simple management. Quite a few corporations (such as Intel, IBM, EMC, NetApp, Mallenox, Brocade, Broadcom, VMware, HuaWei, Cisco, etc.) have released F-CoE SAN related hardware/software solutions. To meet the demands of high-speed data transmission, more IT industries consider high-performance FCoE storage connectivity when upgrading existing IT configurations or building new data centers. And TechNavio [4] reports that the Global FCoE market will grow at a CAGR (Gross Annual Growth Rate) of 37.93% by 2018.

Modern data centers have to handle physical constraints in space and power [1]. These constraints limit the system scale (the number of nodes or servers) when considering the computational density and energy consumption per server [5]. In such cases, improving the scaling-up capacities of system components would be a significant way to alleviate the requirements for servers and system costs. These system capacities include the computing or I/O capacity of individual computation node. Hence, an efficient and scalable stack for accessing remote storage in FCoE-based SAN storage is important to meet the growing demands of users. Moreover, scaling up is well suited to the needs of business-critical applications such as large databases, big data analytics, as well as academic workloads and research.

The software I/O stack suffers from the scaling-up pressure in FCoE-based SAN storage systems with the following features: (1) More cores/CPUs. The availability of powerful, inexpensive multi-core processors can support more instances of multi-threaded applications or virtual machines. And this incurs a large number of I/O requests to remote storage devices. (2) Super high-speed network. The 40 Gbps Ethernet adaptors support the access speed of end nodes in the scale of 40Gbps. (3) Super-high IOPS storage devices. With the increasing number of the connected end nodes, such as mobile and smart devices, data center administrators are inclined to improve the throughput and latency by using the non-volatile memory (NVM) based storage devices. In such cases, software designers need to rethink the importance and role of software in scaling-up storage systems [6], [7], [8].

The Linux FCoE protocol stack (Open-FCoE) is widely used in FCoE-based SAN storage systems. Through experiments and analysis we find that its shared queue & lock mechanism for synchronized accessing the shared queue is apt to lead to a high I/O overhead and limited I/O scalability in multi-core servers. In the Open-FCoE stack, even if we increase the number of cores submitting the 4KB I/Os to a single remote target, the total throughput is no more than 620MB/s in 10Gbps link. This result is only a small fraction of the maximum throughput (around 1200MB/s) in 10Gbps link, thus access bottleneck would worsen in the 40Gbps link.

Lock contention has long been considered as a key impediment to software scalability [9], [10], [11], [12]. Existing works focus on improving the efficiency of lock algorithm (such as [12], [10]) or reducing the number of locks (such

*Corresponding author

as MultiLanes [13] and Tyche [14]) to decrease the synchronization overhead. However, the synchronization problem still exists and leads to a limited scalability. Tyche minimizes the synchronization overhead by reducing the number of synchronization points (spin-locks) to provide scaling with the number of NICs and cores in a server. But Tyche gains less than 2GB/s for 4KB request size with six 10Gbps NICs. Unlike existing solutions, our approach uses private per-CPU structures and disables the kernel preemption [15] to avoid the synchronization overhead. On one hand, each core only accesses its own private per-CPU structures, thus avoiding the concurrent accessing from the threads running in other cores. On the other hand, when the kernel preemption is disabled, the current task (thread) will not be switched out during the period of access to the private structures, thus avoiding the concurrent access from the threads in the same cores. This approach avoids the synchronization overhead. Our scheme achieves 4315.6MB/s throughput with four 10Gbps CNAs for 4KB read requests.

In this paper, we propose FastFCoE, a storage I/O stack for remote storage access in FCoE based SAN storage. FastFCoE is based on the next-generation multi-queue block layer [11], designed by Bjørling and Jens Axboe *et al.*. The multi-queue block layer allows each core to have per-CPU queue for submitting I/O. For further I/O efficiency, FastFCoE directly maps the requests from the block-layer to the FCoE frames. In this way, FastFCoE significantly decreases the I/O process overhead and improves the single core throughput. For instance, when we use one core to submit random 4KB read (write) requests with all FCoE related hardware offload capacities enabled, the throughput of current Open-FCoE stack is 142.25 (216.78)MB/s and the average CPU utilization is 19.65% (13.25%); in contrast, FastFCoE achieves 561.37 (415.39)MB/s and only 15.66% (10.31%) CPU utilization.

Our contributions are summarized as follows:

1. To analyse the performance bottlenecks of FCoE initiator in the existing FCoE-based SAN storage stack, we study the current Open-FCoE stack. In current Open-FCoE stack, each I/O request has to go through several expensive layers, resulting in extra CPU overhead and processing latency. Moreover, in each of SCSI/FCP/FCoE layers, there is a global lock to provide synchronized access to the shared queue in multi-core systems. This shared queue & lock mechanism would lead to happen of LLC cache miss frequently and limited I/O throughput scalability, no more than 21K IOPS.

2. To support an efficient and scalable I/O for remote storage access in the FCoE-based SAN storage in the multi-core servers, we propose a software I/O framework FastFCoE for FCoE initiator. The main features of our design include (i) Improve the parallel I/O capacity. FastFCoE uses the private per-CPU structures for both transmitting and receiving sides and disables the kernel preemption to avoid synchronization overhead. (ii) Shorten I/O path and Reduce Overhead. FastFCoE directly initializes the FCoE frames with I/O requests from the block layer, thus avoiding inter-operations and intra-operations between SCSI/FCP/FCoE layers. In addition, FastFCoE runs under the block layer and supports all upper softwares, such as file systems and applications. FastFCoE call the standard network interfaces. Hence, FastFCoE can use the existing hardware offload features of CNAs (such as scatter/gather I/O, FCoE segmentation offload, CRC offload, FCoE Coalescing and Direct Data Placement offload [16]) and offer flexible use in existing infrastructure (e.g., adaptors, switches, storage devices).

3. We evaluate our FastFCoE, compared with the Open-FCoE stack. Experimental results demonstrate that FastFCoE not only improves single core I/O performance for remote target access performance in FCoE based SAN storage, but also enhances the I/O scalability with the increasing number of cores in multi-core servers. For instance, when using a single thread to submit 64 outstanding 4KB random read/write requests, the throughput of the Open-FCoE is 36408/67006 IOPS, whereas FastFCoE is 143695/106308 IOPS. Further more, to examine the I/O scalability of FastFCoE, we bond four Intel 10Gbps X520 CNAs as a 40Gbps CNA in Initiator and Target servers. FastFCoE can obtain up to 1100K/830K (for 4KB size read/write) IOPS to a remote target and achieves the near maximum throughput for 8KB or larger request sizes.

The remainder of the paper is organized as follows. In Section 2 we review the current implementation of the Linux Open-FCoE protocol stack and analyse its performance bottlenecks detailedly. In Section 3 we propose and detail FastFCoE, an efficient and scalable framework for FCoE protocol stack. Section 4 evaluates the performance and scalability of FastFCoE. We discuss the related work in Section 5 and conclude in Section 6.

II. Open-FCoE

Open-FCoE project [17], the de-facto standard protocol stack for Fibre Channel over Ethernet in different operating systems, is an open-source implementation of an FCoE initiator. Figure 1 shows the layered architecture of Linux Open-FCoE. Each I/O has to traverse several layers from application to hardware. The block layer allows applications to access diverse storage devices in a uniform way and provides the storage device drivers with a single point of entry from all applications, thus alleviating the complexity and diversity of storage devices. The SCSI layer mainly constructs SCSI commands with I/O requests from the block layer. The Libfc (FCP) layer maps SCSI commands to Fibre Channel (FC) frames as defined in standard Fibre Channel Protocol for SCSI (FCP) [18]. The FCoE layer encapsulates FC frames into FCoE frames or de-encapsulates FCoE frames into FC frames as FC-BB-6 standard [3]. The Ethernet driver transmits/receives FCoE frames to/from hardware.

A. Revisiting the I/O process in Open-FCoE

Figure 2 shows the general implementation of the SCSI/FCP/FCoE layers when multiple cores/threads submit I/O requests to the same remote target in multi-core systems. We describe the I/O transmission in Open-FCoE as follows :

1) The SCSI layer builds the SCSI command structure describing the I/O operation from the block layer; then acquires the shared lock before: (1) enqueueing the SCSI command into the shared queue in the SCSI layer;(2) dispatching the SCSI command from the shared queue in the SCSI layer to the FCP layer.
2) The FCP layer builds the internal data structure (FCP request) to describe the SCSI command from

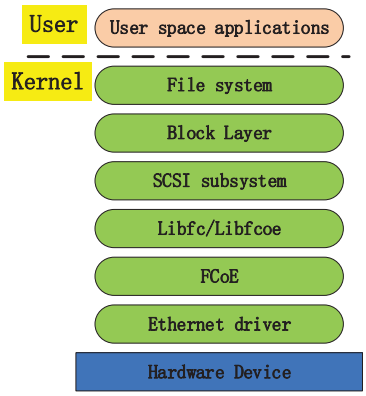


Fig. 1. The architecture of Linux Open-FCoE stack.

the SCSI layer and acquires the shared lock before enqueueing the FCP request into the internal shared queue in the FCP layer. Then, it initializes an FC frame with *sk_buff* structure for the FCP request, and delivers it to the FCoE layer.

3) The FCoE layer encapsulates FC frame into FCoE frame; then acquires the shared lock before: (1) enqueueing the FCoE frame; (2) dequeueing and transmitting the FCoE frame to network device with the standard interface *dev_queue_xmit()*.

There are similar lock acquires in the process of I/O completion, which would significantly impact on the software performance in multi-core systems.

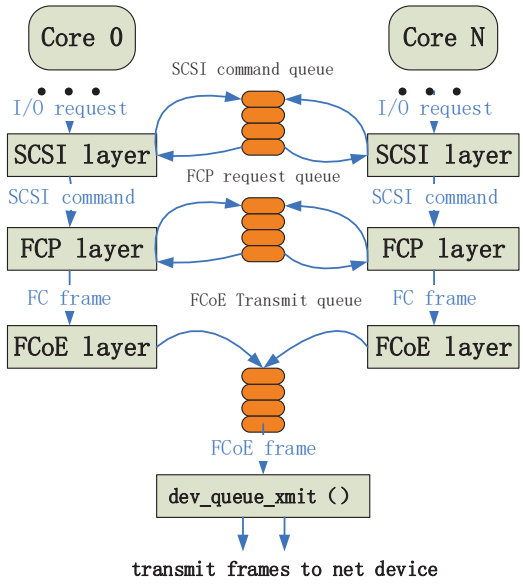


Fig. 2. The process of I/O requests transmission in current Open-FCoE stack.

B. understanding process overhead

1) Single Core Overhead Breakdown: As shown in Figure 1, there are multiple software layers for each I/O to traverse from the block layer to network hardware. This layered architecture in Open-FCoE stack increases the process CPU overhead and latency.

To evaluate how much time is consumed in the process of I/O operation, we have defined some tracepoints for the read request in Linux (kernel 3.13) to investigate the execute time in various layers. Table I shows the path and execution time in different layers when generating an FCoE command frame. The "delta" column indicates the time consumed at each layer.

TABLE I. THE PATH AND EXECUTION TIME FOR ONE I/O REQUEST IN OPEN-FCOE

	Function	**Delta(μs)**
Block layer	Submit_bio	24.235
SCSI layer	Blk_peek_request	22.762
FCP layer	Fc_queuecommand	22.143
FCoE layer	Fcoe_xmit	7.390

As shown in Table I, we observe that the Block/SCSI/FCP layers in the stack consume large fractions of execution time. The execution time proportion of the Block:SCSI:FCP:FCoE layer is 24.235μs:22.762μs:22.143μs:7.390μs. We find several functions in SCSI/FCP/FCoE layers take a long time to translate I/O request into FCoE command frame. For example, the main function of SCSI layer is to allocate and initialize a SCSI command structure with the *request* structure. In the FCP layer, the internal structure (FCP structure) is allocated and initialized with the SCSI command; then allocates the FC format frame and fills the values in its related fields, such as copying the SCSI CDB to the frame. Extra costs are consumed in SCSI/FCP/FCoE layers, such as SCSI command/FCP structure related operations and copying the SCSI CDB to the frame. We classify all the extra overheads into two types, the inter-layer and intra-layer overheads. In this paper we directly initialize the FCoE frame with the I/O request from the block layer. This method cuts the extra inter-layer and intra-layer cost and significantly improves the I/O efficiency (detailed analysis in subsection IV-B).

2) Multi-core Scalability Analysis: For scalability purpose, modern servers employ cache coherent Non Uniform Memory Access (cc-NUMA) in multi-core architecture, such as the one depicted in Figure 3 that corresponds to the evaluation system in this paper. In such architecture, there are some representative features [19], [20], [21], [11], [22], [23], [24], [25] that cause significantly impacts on the software performance, such as Migratory Sharing, False Sharing and significant performance difference when accessing local or remote memory. These features bring challenges to the developers for multi-threaded software in cc-NUMA multi-core systems.

We investigate the I/O scalability of Open-FCoE stack with the mainstream cc-NUMA multi-core architecture. We find that there are bottlenecks not only in the block layer [11] but also in the SCSI/FCP/FCoE layers in terms of the I/O scalability with the increasing number of cores. Specifically, we describe the details of the problems as follows :

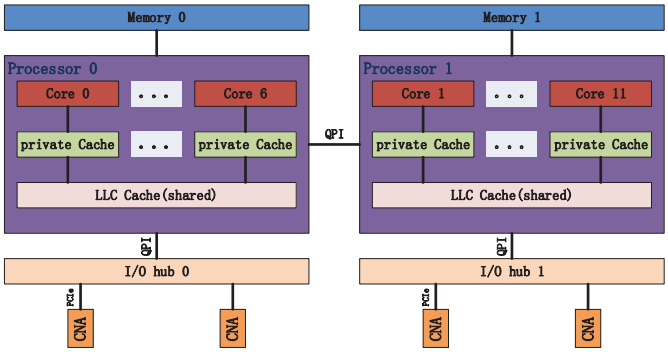
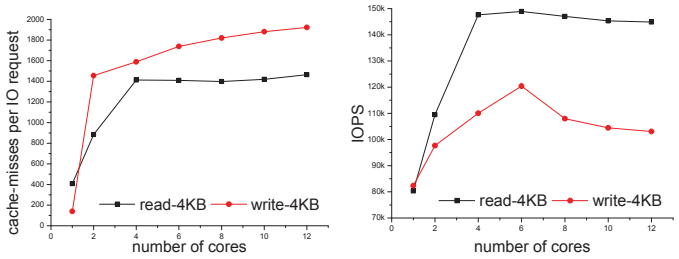


Fig. 3. The multi-core architecture with cache coherent Non-Uniform Memory Access (cc-NUMA).



(a) average LLC cache misses per I/O (b) total throughput

Fig. 4. The average LLC cache misses per I/O and throughput(IOPS) for 4KB random I/O using Open-FCoE stack as a function of number of cores issuing I/Os in a 2 sockets system. The cores are distributed uniformly in a 2 sockets system.

Single queue and global shared lock: As shown in Figure 2, in each layer of the Open-FCoE stack, there is one shared queue and lock. The lock provides coordinated access to the shared data when multiple cores are updating the global queue or list. A high lock contention can slow down the system performance. The more intensive I/Os there are, the more time it consumes to acquire the lock. This bottleneck significantly limits the I/O scalability in multi-core systems.

Migratory sharing: We illustrate this problem with two cases [21]. (1) First, when one or more cores are to privately cache a block in a read-only state, another core requests for writing the block by updating its private cache. In this case, it can lead to incoherence behavior that the cores are caching an old value indefinitely. In the coherence protocol, the shared cache (LLC, Last Layer Cache) forwards the request to all private caches. These private caches invalidate their copies of the block. This increases the load in the interconnection network between the cores and decreases performance when a core is waiting for coherence permissions to access a block. (2) If no other cores cache the block, a request has the negligible overhead of only updating the block in the private cache. Unfortunately, the migratory sharing pattern (i.e. the first case) generally occurs in the shared data access in the current Open-FCoE stack. There are several major sources of migratory sharing patterns in the Open-FCoE stack: (i) shared lock, such as lock/unlock before enqueue/dequeue operations in SCSI/FCP/FCoE layers, exchange memory allocation/free from one global mempool in the FCP layer. (ii) insert or remove the elements from a shared queue or list. Each of SCSI/FCP/FCoE/block layer has one or more shared queues or lists, as shown in Figure 2.

The "false sharing" [21] means that two cores are reading and writing different data on the same cache block respectively. In the remote memory access on NUMA system, the remote cache line invalidation and the large cache directory structures are expensive, thus leading to performance decrease. The shared lock contention, which can frequently result in these problems, will be exacerbated [11] and adds extra access overheads for each I/O in multi-core processors systems.

When multiple cores distributing on different sockets issue intensive I/O requests to a remote target, the shared queue & lock mechanism causes lots of shared data access overheads due to the LLC cache misses and remote memory access. As shown in Figure 4, 4KB I/Os are submitted to a remote target with the current Open-FCoE stack. The average number of cache misses per I/O is depicted in Figure 4(a) as a function of the number of cores that submit I/Os simultaneously. We observe that the average throughput, as shown in Figure 4(b), does not increase too much with the increasing number of cores. But each I/O generates much more average LLC cache misses compared with only one core in Figure 4(a).

III. FastFCoE Design

The analysis in Section 2 shows that current I/O stack has two challenges: (1)How to decrease the overhead for each I/O request? (2)How to improve system scalability in terms of throughput with the increasing number of cores? These problems, which become the bottlenecks in high-performance FCoE-based SAN storage, should be considered along with the evolution of high-performance storage device and high-speed network. In this Section, we present the design of FastFCoE which runs at servers for remote target access in FCoE-based SAN storage. Before we detail our design, we present the design goals and principles.

A. Design Goals

We design the FastFCoE with the following goals:

1) Reduce the process overhead per I/O request and achieve good I/O scalability with the increasing number of cores in multi-core systems.
2) Obtain the efficiency without the cost of decreasing compatibility and flexibility. Fully meet the related standards such as FC-BB-6 [3] and FCP [18]. Use the standard software interfaces and need not to revamp the upper and lower layer softwares. Also a distinctive feature of FastFCoE is simple to use and tightly integrated with existing Linux system without the needs of specific devices or hardware features.

B. Architecture Overview

Figure 5 illustrates the architecture and the overall primary abstractions of FastFCoE. We mainly focus on the I/O subsystem. At the top of the architecture, there are multiple cores that implement the application threads and submit I/O requests to the multi-queue block layer for remote target access. The block layer hides the complexity and diversity of storage devices from the applications while providing common services that

are valuable to applications. Our design is based on the multi-queue block layer [11] that allows each core to have per-CPU queue for submitting I/O.

Our proposed FastFCoE is under the multi-queue block layer and consists of three key components: FCoE Manager, Frame Encapsulation and Frame Process. The key insights include (1) In order to fully leverage parallel I/O capacity with multiple cores, we implement private per-CPU structures (per-CPU variables used by CRC Manager, Exchange Manager, etc.,detailed in Section III-C) to process I/Os instead of the global shared variables accessing, such as single shared queue & lock mechanism. For example, the Exchange Manager uses related per-CPU variables to manage the Exchange ID respectively for each I/O. During the ultra-short period of accessing the per-CPU data, the kernel preemption is disabled and the current Exchange Manager task (thread) will not be switched out. This method avoids the synchronization overhead and significantly improves the parallel I/O capacity. Each core has its own per-CPU structure, thus causing extra spatial overhead for duplicate data in the software layer. Due to the slight spatial overhead (768 Byte private per CPU structures for one core), it has a slight impact on entire system performance. (2) For a shorter I/O path, we directly encapsulate FCoE frames with I/O requests from the block layer and call standard network interfaces to transmit/receive FCoE frames to/from network hardware. This reduces the inter-layer and intra-layer overheads from SCSI/FCP/FCoE layers.

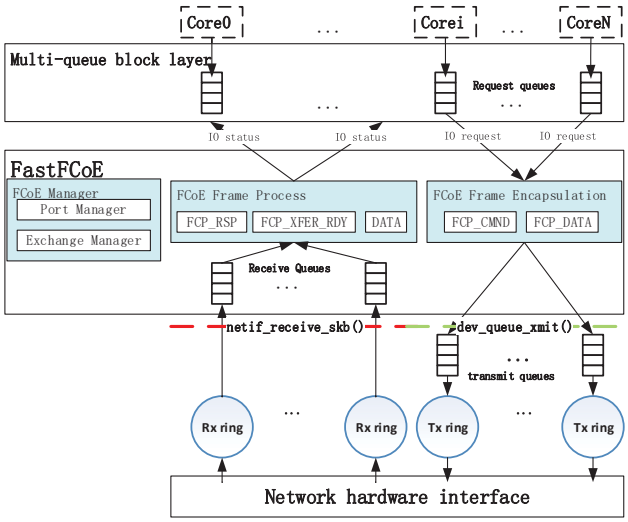


Fig. 5. The FastFCoE architecture in multi-core server. Note: The remote FCoE SAN storage target is mapped as a block device.

The network link layer is under the FastFCoE. The frames from FastFCoE is transmitted to the network device (C-NA, converged network adaptor) by the standard interface *dev_queue_xmit()*. The standard interface *netif_receive_skb()* processes the received frames from network. All the hardware complexity and diversity of CNAs are transparent to FastFCoE.

In addition, almost all modern converged network adaptors have multiple hardware Tx/Rx queues to enable transmitting or receiving parallelization, as shown in Figure 5. For instance, the Intel X520 10GbE converged network adaptor has 128 Tx queues and Rx queues.

C. Design Components

FastFCoE provides a low process overhead and high I/O throughput scalability for remote target access in FCoE-based SAN storage, which mainly consists of the FCoE Manager, the FCoE Frame Encapsulation, and the FCoE Frame Process. Other functions have no impacts on I/O performance and are overlooked in this paper, such as fabric login/logout, N_Port login/logout and Fibre Channel link services, etc. as defined in FCP [18] or FCoE standards [3].

1) FCoE Manager: The FCoE Manager has two components: (1) the Port Manager, which mainly manages the states or information of the local port and remote port etc. (2) the Exchange Manager, which mainly manages the allocation/free of Exchange and Sequence ID. A sequence is a set of one or more related Frames transmitted unidirectionally from one end Node to another end Node. An Exchange consists of one or more unidirectional Sequences, initiated by either the Exchange Originator or the Exchange Responder. Each FCoE/FCP I/O request is identified by the initiator address identifier, the target address identifier, the Exchange Identifier for the Originator (OX_ID) and the Exchange Identifier for the Responder (RX_ID). All these field values are provided by the Port Manager and the Exchange Manager.

Port Manager. All the information or states of the Fabric, the Enode[1], the Virtual Links[2] and VN_Ports[3], etc. are managed and maintained by the Port Manager. The Port Manager can provide the related field values to initialize the transmitted FCoE frames and decide the received FCoE frames to implement the I/O operations for remote storage access.

Exchange Manager. Each I/O operation defined by Fibre Channel Protocol for SCSI (FCP) standard [18] is mapped into an Exchange. For each I/O operation, the Enode (Initiator) originates an Exchange and assigns a unique Originator Exchange ID (called OX_ID). When the Responder Enode (Target) receives the first Sequence of the Exchange, it assigns a Responder Exchange ID (RX_ID) to the newly established Exchange. As mentioned above, the allocation efficiency of Exchange ID is very important to the performance of each I/O request.

FastFCoE allocates/frees the Exchange ID for each I/O request by this Exchange Manager. To allocate/free the Exchange ID efficiently, we use the simple and useful method that all values of Exchange IDs are distributed in each core. The main structure of the Exchange Manager is consisted of a pre-allocated and per-CPU array, as shown in Figure 6. Each element in array associates with an Exchange ID value and a pointer. If the pointer points to a request, this Exchange ID has been used by an Exchange. Otherwise if the pointer is NULL, this Exchange ID can be used for a new Exchange. To avoid the synchronization overhead, we also disable the kernel preemption to allocate/free the Exchange ID one by

[1]FCoE Node is called ENode, such as FCoE Initiator and Target.

[2]FCoE Virtual Links replace the physical Fibre Channel links by encapsulating FC frames in Ethernet frames.

[3]A device virtual port that generates/terminates FC traffic is called VN_Port.

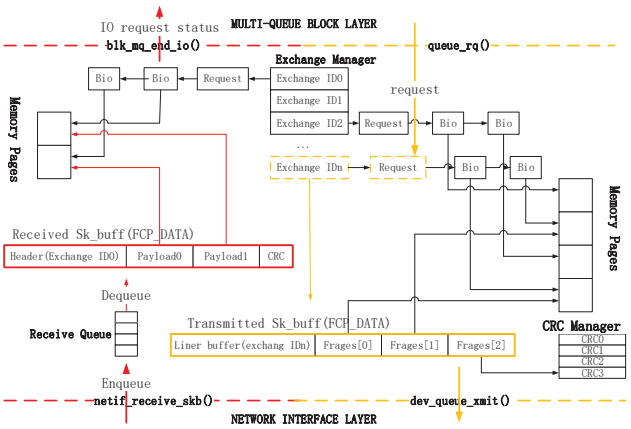


Fig. 6. The structures in FastFCoE in transmitting and receiving the FCoE FCP_DATA frames(sk_buff).

one by changing the pointer value, which is also a per-CPU variable.

2) FCoE Frame Encapsulation: FCoE Frame Encapsulation implements different types of FCoE frames manufacture and transmission, mainly including FCP_CMND (representing the data delivery request) and FCP_DATA (representing the data delivery) frames at Initiator.

As shown in Figure 6, the I/O request from the block layer consists of several segments, which are contiguous on the block device, but not necessarily contiguous in physical memory, depicting the mapping between a block device sector region and some individual memory segments. Hence, the FCP_DATA frame payloads (the transferred data) are not contiguous in physical memory and the length of FCP_DATA frame payloads is almost larger than the FCoE standard MTU(adapter maximum transmission unit). On the other hand, the hardware function, scatter/gather I/O, directly transfers the multiple non-liner memory segments to the hardware by DMA. In addition, FCoE segmentation offload (FSO) [16] is a technique for increasing the outbound throughput of high-bandwidth converged network adaptor (CNA) by reducing CPU overhead and it works by queueing up large frames and allowing the hardware (CNA) to be split into multiple FCoE frames. To reduce the overhead and support these hardware capacities, we use the linear buffer of the *sk_buff* structure to represent the header of FCoE FCP_DATA frame and the *skb_shared_info* structure to point to these non-linear buffers to present a large transferred data. These non-linear buffers include request segments in memory page and the CRC, EOF (not shown in Figure 6) fields in FCP_DATA frame. In addition, to improve system efficiency, we use the pre-allocation method that obtains a special memory page to manage the CRC and EOF allocation for each core.

The FCoE FCP_CMND frame encapsulation is similar with FCP_DATA frame, but only uses the linear buffer of the sk_buff structure to depict the frame. Moreover, the FCoE Frame Encapsulation also supports the hardware Direct Data Placement offload (DDP) [16] capacity which reduces CPU overhead by directly tranfering the FCP_DATA frame payload to the request memory segments. For read requests, the FCoE Frame Encapsulation sets up the function of hardware DDP offload by standard interface *ndo_fcoe_ddp_setup()*.

Rather than using several layers to handle and queue the I/O requests, the FCoE Frame Encapsulation quickly allocates and initializes the FCP_CMND frames with the information from the FCoE Manager and the I/O requests. To avoid queue operations or other extra overheads, the FCoE Frame Encapsulation component does not have any queues and directly sends the frames to the lower layer using the common network interface *dev_queue_xmit()*. These schemes improve Cores/CPUs parallel I/O capacity and reduce the overhead with a shorter I/O path.

3) FCoE Frame Process: The main function of FCoE Frame Process buffers and deals with the received FCoE frames. There is a private per-CPU queue for each core, called Receive Queues as shown in Figure 6. All the received FCoE frames are mapped into the corresponding private queue by the OX_ID field. For each private per-CPU queue, FastFCoE uses a spin-lock for synchronization when buffering the received FCoE frames. By this way multiple locks are distributed on different per-CPU queues, thus significantly reducing the intensity of lock contention and the synchronization overheads. This per-CPU receive queue design improves parallel I/O capacity in multi-core systems. All FCoE frames from Receive Queues are processed, as defined in FCP standard [18].

Our design has low process overhead and high throughput to perform for both read and write operations. It uses the standard interfaces and needs not to revamp the upper and lower layers. Moreover, the hardware capacities, such as scatter/gather I/O, FCoE segmentation offload (FSO), CRC offload, FCoE Coalescing and Direct Data Placement offload (DDP) [16], can be used as usual.

IV. Experimental Evaluation

In modern data centers, there are two common deployment solutions for servers, including traditional non-virtualized server (a physical machine) and virtualized server (a virtual machine). In this section, we use experimental results to answer the following questions under both non-virtualized and virtualized systems: (1) Does FastFCoE consume less process overhead per I/O request than standard Open-FCoE stack? (2) Does FastFCoE achieve better I/O scalability with the increasing number of cores on multi-core platform? (3) How is the performance of FastFCoE influenced under different degrees of CPU loads? Before answering these questions, we describe the experimental environment.

A. Experimental Method and Setup

To understand the basic aspects of our FastFCoE, we evaluate the main features with two micro-benchmark FIO [26] and Orion [27]. FIO is a flexible workload generator. Orion is designed for simulating Oracle database I/O workloads and uses the same I/O software stack as Oracle databases.

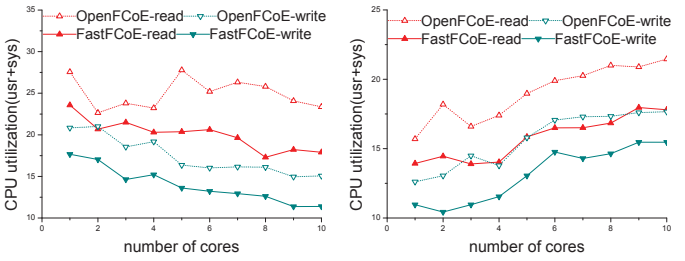
In addition, we analyze the impact of throughput performance under different degrees of CPU loads with real world TPC-C [28] and TPC-E [29] benchmark traces, which are used extensively for evaluating OLTP performance by industry and research community.

We use the Open-FCoE stack in the Linux kernel as baseline to carry out the comparisons. Our experimental platform consists of two systems (initiator and target), connected back-to-back with multiple CNAs. Both initiator server and target server are configured with Dell PowerEdge R720, Dual Intel Xeon Processor E5-2630 (6 cores, 15MB Cache, 2.30 GHz, 7.20 GT/s Intel QPI), 64GB DDR3, Intel X520 10Gbps CNAs, with hyperthreading capabilities enabled. The Open-FCoE or FastFCoE stack runs in the host or virtual machines with CentOS 7 (3.13.9 kernel). The target system is based on the modified Linux I/O target (LIO) 4.0 with CentOS 7 (3.14.0 kernel) and uses 40GB ram as a disk. Note that we use ram based disk and back-to-back connection only to avoid the influences from network and slow target system. Hardware Direct Data Placement offload (DDP) [21], the hardware offload functions for FCoE protocol, is enabled when the request size is equal to or larger than 4KB.

B. Performance Results

First, we compare FastFCoE with Open-FCoE in terms of the average CPU overhead and latency by sending "simple/meaningless/no real data" I/O requests. Then, We evaluate the I/O scalability with the increasing number of concurrent I/Os using Orion and the I/O scalability with the increasing number of cores submitting I/Os using FIO. Finally, we use two benchmark traces (TPC-C and TPC-E) to evaluate throughput performance between FastFCoE and the Open-FCoE under different degrees of CPU loads.

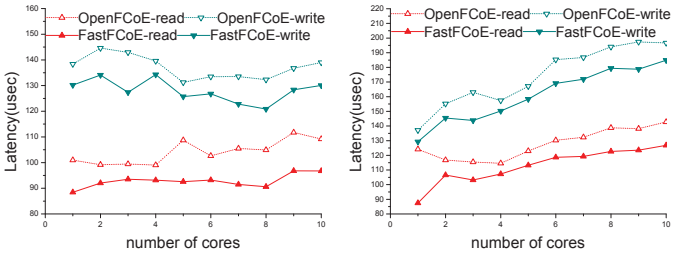
1) CPU usage and Latency Evaluation: The CPU usage and latency are measured by issuing a single outstanding I/O per participating cores at a time and using the libaio interface of the Linux kernel in the non-virtualized and virtualized systems with 10Gbps CNA respectively.



(a) CPU utilization in the non-virtualized system (b) CPU utilization in the virtualized system

Fig. 7. The CPU utilization evaluation with FIO. The CPU utilization of FastFCoE and Open-FCoE, by issuing a 512B size single outstanding I/O per participating core at a time.

The CPU usage is the system time utilization plus user time utilization. Due to the layered architecture, Open-FCoE has to traverse SCSI/FCP/FCoE layers, which results in the extra inter-operations and intra-operations between SCSI/FCP/FCoE layers, whereas FastFCoE avoids those extra CPU overheads. As a result, FastFCoE has less CPU overhead for each I/O request than Open-FCoE. Figure 7 shows the average CPU utilization depending on the number of cores. For the non-virtualized system, the average CPU utilization of FastFCoE has a decrease of 1.97%∼8.5% and 2.8%∼3.98% for read and write respectively. For the virtualized system, the average CPU utilization of FastFCoE has a decrease of 1.77%∼4.16% and 1.65%∼3.53% for read and write respectively. The hareware capacity of DDP is disabled in 512B read operation, thus requiring higher CPU overhead than write operation.



(a) The average latency in the non-virtualized system (b) The average latency in the virtualized system

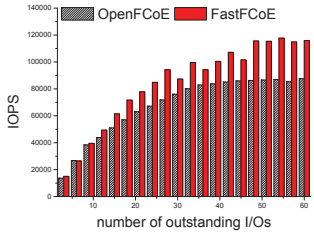
Fig. 8. The latency of FastFCoE and Open-FCoE, by issuing a 512B size single outstanding I/O per participating core at a time.

The latency is measured as the time from the application, through the kernel, into the network. Our FastFCoE has a short I/O path. Hence, FastFCoE has a smaller average latency than Open-FCoE. Figure 8 shows the average latency depending on the number of cores. For the non-virtualzied system, the average latency of FastFCoE has a decrease of 5.92∼16.14 and 5.33∼15.58 microseconds for read and write request respectively. For the virtualized system, the average latency of FastFCoE has a decrease of 7.33∼36.68 and 7.12∼19.17 microseconds for read and write request respectively. The write operation causes higher complexity in FCP protocol [18] than read operation. Therefore, the write operation has a larger latency than read operation.

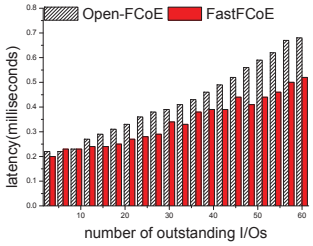
2) I/O scalability Evaluation: The improvements of the I/O scalability with the increasing number of concurrent I/Os and the increasing number of cores submitting I/Os are important to I/O subsystem. In this subsection, we use FIO and Orion [27] to evaluate the I/O scalability of FastFCoE in the non-virtualized and virtualized systems respectively.

We use a single Orion instance to simulate OLTP (Online transaction processing) and DSS (Decission support system) application scenarios. OLTP applications generate small random reads and writes, typically 8KB. Such applications usually pay more attention to the throughput in I/Os Per Second(IOPS) and the average latency (I/O turn-around time) per request. These parameters directly determine the transaction rate and transaction turn-around time at the application layer. DSS applications generate random 1MB I/Os, stripped over several disks. Such applications process large amounts of data, and typically examine the overall data throughput in MegaBytes per second(MB/S).

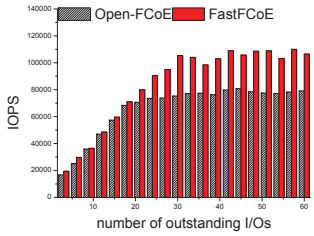
We evaluate the performance in OLTP (as shown in Figure 9) and DSS (as shown in Figure 10) application scenarios with 50% write requests on FastFCoE and Open-FCoE in 10Gbps Ethernet link respectively. With the increasing number of concurrent I/Os, the I/Os become more intensive. Since FastFCoE has better scalability than Open-FCoE in both non-virtualied and virtualied systems, the performance gap in terms of throughput and latency becomes larger when using more concurrent I/Os. For OLTP, the average throughput (IOPS)



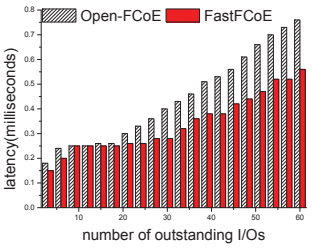
(a) IOPS for OLTP in the non-virtualized system



(b) Latency for OLTP in the non-virtualized system

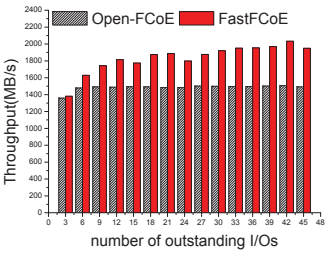


(c) IOPS for OLTP in the virtualized system

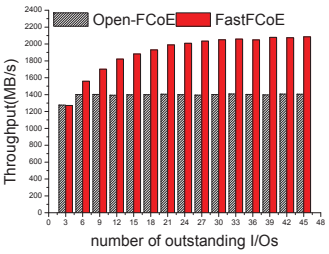


(d) Latency for OLTP in the virtualized system

Fig. 9. I/O scalability Evaluation with Orion (50% write). The Figure shows the average throughput and latency obtained by FastFCoE and Open-FCoE in different numbers of outstanding IOs for OLTP test, with the non-virtualized and virtualized systems respectively.



(a) Throughput for DSS in the non-virtualized system



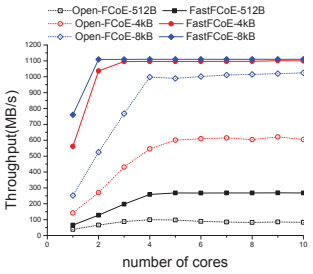
(b) Throughput for DSS in the virtualized system

Fig. 10. I/O scalability Evaluation with Orion(50% write). The figure shows the average throughput obtained by FastFCoE and Open-FCoE in different numbers of outstanding IOs for DSS test, with the non-virtualized and virtualized systems respectively.

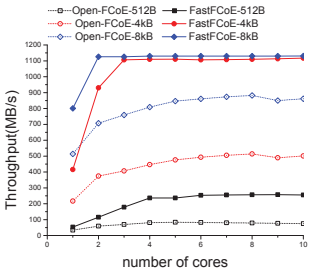
of FastFCoE outperforms Open-FCoE by 1.35X and 1.28X at most, in the non-virtualied and virtualied system. At the same time the average latencies have 26.8% and 28.9% reduction respectively. For DSS, the throughput of FastFCoE outperforms Open-FCoE by 1.350X and 1.489X at most, in the non-virtualied and the virtualied system. This is because that FastFCoE has smaller process overheads than Open-FCoE.

One challenge for I/O stack is the limited I/O scalability for small size requests in multi-core systems [14]. To show the scalability behavior for small size requests, we use FIO to evaluate the I/O scalability with the increasing number of cores submitting I/Os. We set the permitted number of cores with 100% utility and bind one thread for each permitted core.

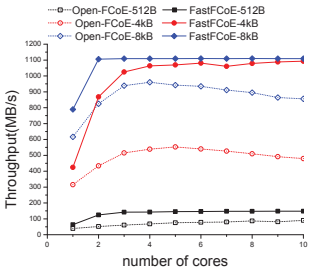
Figure 11 shows the total throughput by submitting 64 outstanding asynchronous random 512B,4KB and 8KB size requests with different numbers of cores with 10Gbps CNA. For



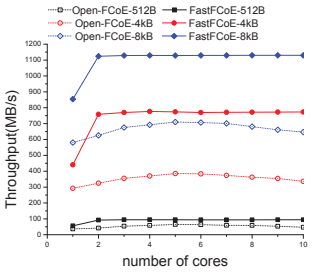
(a) the non-virtualized system read



(b) the non-virtualized system write



(c) the virtualized system read



(d) the virtualized system write

Fig. 11. Scalability Evaluation with FIO (random workload). The figure shows the total throughput of FastFCoE and Open-FCoE when changing the number of cores submitting 64 outstanding 512,4KB and 8KB I/O requests in the non-virtualized and virtualized systems with 10Gbps CNA.

the non-virtualized system, when using one core, our FastFCoE shows higher throughput than Open-FCoE by 1.71/1.58X, 3.95/1.92X and 3.02/1.56X on 512B, 4KB, 8KB read/write requests. For 512B read requests, FastFCoE achieves almost the highest throughput of a single CNA, 529011 IOPS(258.34MB/s) with only 4 cores, whereas the Open-FCoE is 204074(99.66MB/s). This shows that Open-FCoE has the limited throughput (IOPS), about 20K. The non-virtualized system has a better throughput than the virtualized system. For 4KB and 8KB requests, the non-virtualized system can achieve near maximum throughput in 10Gbps link with 2 or 3 cores. For the virtualized system, when using one core, FastFCoE gets higher throughput than Open-FCoE by 1.63/1.53X, 1.35/1.51X and 1.28/1.47X on 512B, 4kB, 8kB read/write requests. For 512B read/write requests, FastFCoE achieves 302970/191881 IOPS (148.46/93.69MB/s) at most, whereas the Open-FCoE is 184879/132223 IOPS (90.28/64.56MB/s). Our approach in FastFCoE uses the private per-CPU sturctures on both transmitting and receiving sides and disables kernel preemption to avoid synchronization overhead. This approach significantly improves the I/O scalability with the number of cores.

To further study the I/O scalability of FastFCoE, while avoiding the influence from limited capacity of adapter (CNA), we bond four Intel X520 10Gbps CNAs for both the Initiator (non-virtualized server) and Target, running as a single 40Gbps ethernet CNA for the upper layers. The throughput results show that FastFCoE has quite good I/O scalability capacity, as shown in Figure 12. For 4KB read requests, the IOPS of FastFCoE can improve with the increasing number of cores to submit request until around 1.1M IOPS(4315.6MB/s). All though the write operation has higher complexity in FCP protocol [18] than read operation, for 4KB random write,

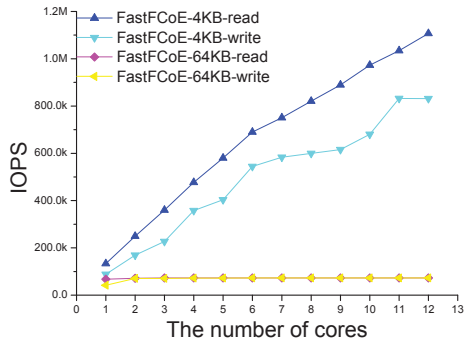
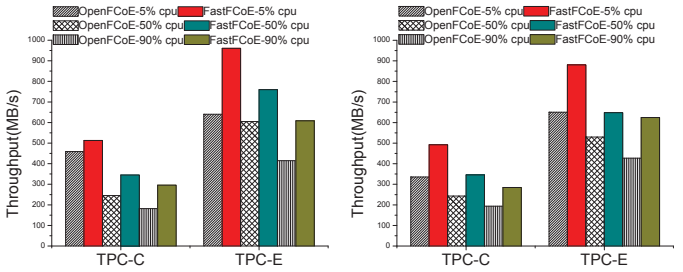


Fig. 12. Scalability Evaluation with FIO in 40Gbps link. IOPS obtained by FastFCoE depends on the number of cores with 4KB and 64KB random read/write request when bonding four 10Gbps CNAs as one 40Gbps CNA in non-virtualization system.



(a) Throughput in the non-virtualized system (b) Throughput in the virtualized system

Fig. 13. Throughput Evaluation with TPC-C and TPC-E. The Figure shows the throughputs achieved by FastFCoE and Open-FCoE, with 5%/50%/90% CUP loads in the non-virtualized and virtualized systems respectively.

FastFCoE still achieves up to 831250 IOPS(4059.3MB/s).

Since I/O stack usually exhibits higher throughput with larger request size [14], for larger size requests, FastFCoE can achieve the higher throughput with less number of cores. With FIO using one thread, FastFCoE obtains 4392.6MB/s for 64KB random read requests. FastFCoE hence has sufficient capacity to fit with 40Gbps link in the FCoE-based SAN storage.

3) TPC-C and TPC-E tests Using OLTP Disk Traces: Many applications consume a large amount of CPU resource and affect I/O subsystem. To show the throughput of FastFCoE over Open-FCoE under different degrees of CPU loads, we analyze the throughput in both the non-virtualized and virtualized systems with a 10Gbps CNA by using OLTP benchmark traces: TPC-C and TPC-E. These traces are obtained from test using HammerDB [30] with Mysql Database and collected at Microsoftware [31]. TPC-E is more read intensive with a 9.7:1 read-to-write ratio I/O, while TPC-C shows a 1.9:1 read-to-write ratio; and the I/O access pattern of TPC-E is random like TPC-C.

The specified loads are generated by FIO tool. We simplely use 5%/50%/90% CPU loads to represent three degrees of CPU loads. To compare the throughput under the same environment, we replay these workloads with the same time stamps within the trace logs. Figure 13 shows the superiority of FastFCoE over Open-FCoE in both the non-virtualized and virtualized systems. The average throughput degrades with the increasing CPU loads for both the TPC-C and TPC-E benchmarks. For the TPC-C benchmark, FastFCoE outperforms Open-FCoE by 1.127X/1.410X/1.629X and 1.467X/1.424X/1.471X in the non-virtualized and virtualized systems with 5%/50%/90% CPU loads respectively. For TPC-E benchmark, FastFCoE outperforms Open-FCoE by 1.500X/1.256X/1.468X and 1.353X/1.224X/1.463X in the non-virtualized and virtualized systems with 5%/50%/90% CPU loads respectively.

V. Related Work

The Scalability on Multi-core Systems. Over the last few years, a number of studies have attempted to improve the scalability of operating systems in current multi-core systems. The lock contention is regarded as one of primary reasons for poor scalability [9], [10], [11], [12]. HaLock [10] is a hardware assisted lock profiling mechanism which leverages a specific hardware memory tracing tool to record the large amount of profiling data with negligible overhead and impact on even large scale multithreaded programs. RCL [12] is a lock algorithm that aims to improve the performance of critical sections in legacy applications on multi-core architectures. MultiLanes [13] builds an isolated I/O stack on top of virtualized storage devices for each VE to eliminate contention on kernel data structures and locks between them, thus scaling them to many cores. Gonzlez-Frez *et al.*[14] present Tyche, a network storage protocol directly on top of Ethernet. It minimizes the synchronization overheads by reducing the number of spin-locks to provide scaling with the number of NICs and cores.

In this paper, to provide a scalable I/O stack, we reasonably use the private per-CPU structures and disable kernel preemption to process I/Os. This method avoids lock contention for synchronization, which significantly decreases the performance scalability in multi-core servers.

Methods for Optimizing I/O Overhead. Software overhead from high-speed I/O obtain a lot of attentions, which consumes substantial system resources and influences on the system performance [32]. Le, Duy and Huang, Hai *et al.* [33] have shown that the choice of the nested file systems on both hypervisor and guest levels has the significant performance impact on I/O performance in the virtualized environments. Jisoo Yang *et al.* [7] show that when using NVM device polling for the completion delivers higher performance than traditional interrupt-driven I/O. Our design of FastFCoE focuses on the issues at the software interface between the host and the CNA, which emerges as an important bottleneck in high-performance FCoE based SAN storage.

To optimize the I/O performance, much work removes the I/O bottlenecks by replacing multiple layers with one flat or a pass-through layer in certain cases. Caulfield *et al.* [6] propose to bypass the block layer and implement their own driver and single queue mechanism to improve I/O performance. In addition, Rizzo and Luigi [34], [35] propose *netmap*, a framework that shows user-pace applications to exchange raw packets with the network adapter, thus making a single core running at 900 MHz to send or receive 14.88Mpps (the peak packet rate on in 10Gbps links). Our FastFCoE is under the block layer and calls the standard network interfaces to transmit/receive network packets. Therefore, FastFCoE can

support all upper softwares (such as existing file systems and applications) and be deployed with existing infrastructure (adaptors, switches, storage devices), without the costs of extra hardware.

In addition, Bjørling and Jens Axboe *et al.* [11] demonstrate that in multi-core systems the single-queue block layer becomes the bottleneck and design the next-generation multi-queue block layer. FastFCoE is the first work to introduce the multi-queue block layer to FCoE protocol process. And it is also the first work to shorten the I/O path by directly mapping the requests from the block-layer to the FCoE frames.

VI. Conclusion

In the context of high-speed network and fast storage technologies, the current FCoE I/O stack becomes a bottleneck, thus leading to a high I/O overhead and limited I/O scalability in multi-core servers. In this work, we present the design, implementation, and evaluation of FastFCoE, a storage I/O stack for remote storage access in FCoE based SAN storage. To improve the multi-core parallel I/O capacity, FastFCoE uses the private per-CPU structures and disables kernel preemption to process I/Os. This method avoids synchronization overhead. Further more, to reduce the I/O process overhead, FastFCoE directly maps the I/O requests from the multi-queue block layer to FCoE frames and calls the standard software interfaces without the needs of extra hardware. Our results show that FastFCoE achieves an efficient and scalable I/O throughput.

Acknowledgment

This work was supported by the National Basic Research 973 Program of China under Grant No. 2011CB302301; 863 Project No. 2013AA013203, No. 2015AA015301, No. 2015AA016701; NSFC No. 61173043, No.61303046, No. 61472153; This work was also supported by Key Laboratory of Information Storage System, Ministry of Education, China.

References

[1] J. Jiang and C. DeSanti, "The role of fcoe in i/o consolidation," in *Proceedings of the International Conference on Advanced Infocomm Technology*. ACM, 2008.

[2] C. DeSanti and J. Jiang, "Fcoe in perspective," in *Proceedings of the International Conference on Advanced Infocomm Technology*. ACM, 2008.

[3] INCITS Project T11.3/2159-D, "Fibre Channel-Backbone-6 (FC-BB-6)."

[4] TechNavio, "Global Fiber Channel over Ethernet Market 2014-2018."

[5] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 37–48, 2012.

[6] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing safe, user space access to fast, solid state disks," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 387–400, 2012.

[7] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt." in *USENIX Conference on File and Storage Technologies*, 2012.

[8] W. Shin, Q. Chen, M. Oh, H. Eom, and H. Y. Yeom, "Os i/o path optimizations for flash solid-state drives," in *USENIX conference on USENIX Annual Technical Conference*, 2014, pp. 483–488.

[9] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, N. Zeldovich *et al.*, "An analysis of linux scalability to many cores." in *OSDI*, vol. 10, no. 13, 2010, pp. 86–93.

[10] Y. Huang, Z. Cui, L. Chen, W. Zhang, Y. Bao, and M. Chen, "Halock: hardware-assisted lock contention detection in multithreaded applications," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 253–262.

[11] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block io: Introducing multi-queue ssd access on multi-core systems," in *Proceedings of the 6th International Systems and Storage Conference*. ACM, 2013, p. 22.

[12] J.-P. Lozi, F. David, G. Thomas, J. L. Lawall, G. Muller *et al.*, "Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications." in *USENIX Annual Technical Conference*, 2012, pp. 65–76.

[13] J. Kang, B. Zhang, T. Wo, C. Hu, and J. Huai, "Multilanes: providing virtualized storage for os-level virtualization on many cores." in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, 2014, pp. 317–329.

[14] P. González-Férez and A. Bilas, "Tyche: An efficient ethernet-based protocol for converged networked storage," in *IEEE Conference on Mass Storage Systems and Technologies*, 2014.

[15] R. Love, *Linux Kernel Development*. Pearson Education, 2010.

[16] Intel Corporation, "Intel 82599 10 Gigabit Ethernet Controller Datasheet," 2012.

[17] Open-FCoE. http://www.open-fcoe.org.

[18] INCITS Project T11.3/1144-D, "Fibre Channel Protocol for SCSI (FCP)."

[19] M. M. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Communications of the ACM*, vol. 55, no. 7, pp. 78–89, 2012.

[20] M. Lis, K. S. Shim, M. H. Cho, and S. Devadas, "Memory coherence in the age of multicores," in *International Conference on Computer Design (ICCD)*. IEEE, 2011, pp. 1–8.

[21] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011.

[22] D. Zhan, H. Jiang, and S. C. Seth, "Stem: Spatiotemporal management of capacity for intra-core last level caches," in *Proceedings of 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2010, pp. 163–174.

[23] D. Zhan, H. Jiang, and S. Seth, "Clu: Co-optimizing locality and utility in thread-aware capacity management for shared last level caches," *IEEE Transactions on Computers*, vol. 63, no. 7, pp. 1656–1667, 2014.

[24] D. Zhan, H. Jiang, and S. C. Seth, "Locality & utility co-optimization for practical capacity management of shared last level caches," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 279–290.

[25] Y. Hua, X. Liu, and D. Feng, "Mercury: a scalable and similarity-aware scheme in multi-level cache hierarchy," in *Proceedings of International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 2012, pp. 371–378.

[26] Flexible io generator. http://freecode.com/projects/fio.

[27] ORION, " ORION: Oracle I/O Numbers Calibration Tool."

[28] TPC-C specification. http://www.tpc.org/tpcc/default.asp.

[29] TPC-E specification. http://www.tpc.org/tpce/default.asp.

[30] HammerDB. http://www.hammerdb.com/index.html.

[31] Microsoft Enterprise Traces. http://iotta.snia.org.

[32] B. H. Leitao, "Tuning 10gb network cards on linux," in *Proceedings of the Linux Symposium*, 2009.

[33] D. Le, H. Huang, and H. Wang, "Understanding performance implications of nested file systems in a virtualized environment." in *USENIX Conference on File and Storage Technologies*, 2012, p. 8.

[34] L. Rizzo, "netmap: A novel framework for fast packet i/o." in *USENIX Annual Technical Conference*, 2012, pp. 101–112.

[35] L. Rizzo and M. Landi, "netmap: memory mapped access to network devices," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 422–423.