I/O Stack Optimization for Efficient and Scalable Access in FCoE-based SAN Storage

Yunxiang Wu, Fang Wang, Yu Hua, *Senior Member, IEEE*, Dan Feng, *Member, IEEE*, Yuchong Hu, Wei Tong, Jingning Liu, Dan He

Abstract—Due to the high complexity in software hierarchy and the shared queue & lock mechanism for synchronized access, existing I/O stack for accessing the FCoE based SAN storage becomes a performance bottleneck, thus leading to a high I/O overhead and limited scalability in multi-core servers. In order to address this performance bottleneck, we propose a synergetic and efficient solution that consists of three optimization strategies for accessing the FCoE based SAN storage: (1) We use private per-CPU structures and disabling kernel preemption method to process I/Os, which significantly improves the performance of parallel I/O in multi-core servers; (2) We directly map the requests from the block-layer to the FCoE frames, which efficiently translates I/O requests into network messages; (3) We adopt a low latency I/O completion scheme, which substantially reduces the I/O completion latency. We have implemented a prototype (called FastFCoE, a protocol stack for accessing the FCoE based SAN storage). Experimental results demonstrate that FastFCoE achieves efficient and scalable I/O throughput, obtaining 1132.1K/836K IOPS (6.6/5.4 times as much as original Linux Open-FCoE stack) for read/write requests.

Index Terms—Storage architecture, Fiber Channel over Ethernet, Multi-core framework.

1 Introduction

In order to increase multi-core hardware utilization and reduce the total cost of ownership (TCO), many consolidation schemes have been widely used, such as server consolidation via virtual machine technologies and I/O consolidation via converged network adapters (CNAs, 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. Many corporations (such as Intel, IBM, EMC, NetApp, Mallenox, Brocade, Broadcom, VMware, HuaWei, Cisco, etc.) have released FCoE 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. 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 cost-efficient way. 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 storage I/O stack suffers from the scaling-up pressure in FCoE-based SAN storage systems with the following features : (1) More cores. The availability of powerful, inexpensive multi-core processors can support more instances of multi-threaded applications or virtual machines. This incurs a large number of I/O requests to remote storage devices. (2) Super high-speed network. The 40Gbps 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 observe that Open-FCoE has a high I/O overhead and limited I/O scalability for accessing the FCoE based SAN storage in multi-core servers. For example, with 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 625MB/s. This result is only a small fraction of the maximum throughput (around

• *Y. Wu, F. Wang, Y. Hua, D. Feng, Y. Hu, W. Tong, J. Liu, D. He are with the Wuhan National Lab for Optoelectronics, Key Laboratory of Data Storage Systems, Ministry of Education of China, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074 China.*
E-mail:{yxwu, wangfang, csyhua, dfeng, yuchonghu, tongwei, jnliu, hdnchu}@hust.edu.cn.

The preliminary version appears in the Proceedings of the 44th International Conference on Parallel Processing (ICPP), 2015, pages 330-339.

1,200MB/s) in 10Gbps link. Access bottleneck would worsen in the 40Gbps link due to the limited I/O scalability in the current Open-FCoE stack.

Lock contention has been considered as a key impediment to improve system scalability [9], [10], [11], [12]. Existing works focus on improving the efficiency of lock algorithm (such as [10] and [12]) or reducing the number of locks (such 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, we uses private per-CPU structures & disabling the kernel preemption [15] method to avoid the synchronization overhead. 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 4,383.3MB/s throughput with four 10Gbps CNAs for 4KB read requests.

In this paper, we introduce a synergetic and efficient solution that consists of three optimization schemes. We have implemented a prototype (called FastFCoE, a protocol stack for accessing the 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 a per-core queue for submitting I/O. For further I/O efficiency, FastFCoE has a short I/O path both on the I/O issuing side and I/O completion side. 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 the current Open-FCoE stack is 142.25 (216.78) MB/s and the average CPU utilization is 19.65% (13.25%), whereas FastFCoE achieves 561.37 (415.39) MB/s throughput and 15.66% (10.31%) CPU utilization.

Our contributions are summarized as follows:

1. We expose the three limitations of the current Open-FCoE stack, which become I/O performance bottlenecks. In the current Open-FCoE stack, (1) each I/O request has to go through several expensive layers to translate the I/O request to network frame, resulting in extra CPU overhead and processing latency. (2) 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 the occurrence of LLC cache miss frequently and limited I/O throughput scalability, no more than 220K IOPS. (3) In the I/O completion path there are at least three context switchings (doing the I/O completion work in FCP/SCSI/BLOCK layer) to inform the I/O-issuing thread of I/O completion. This can lead to additional task scheduling and process overhead.

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 three optimization strategies : (1) We use private per-CPU structures & disabling the kernel preemption method to process I/Os, which significantly improves the performance of parallel I/O in multi-core servers; (2) We directly map the requests from the block-layer to the FCoE frames, which efficiently translates I/O requests into network messages; (3) We adopt a low latency I/O completion scheme, which substantially reduces the I/O completion latency. We have implemented a prototype (called FastFCoE). FastFCoE runs under the block layer and supports all upper software components, such as file systems and applications. Moreover, FastFCoE calls 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 infrastructures (e.g., adaptors, switches and storage devices).

3. We evaluate the three optimization schemes within FastFCoE, compared with the Open-FCoE stack. Experimental results demonstrate that FastFCoE not only improves single core I/O 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 I/Os, the throughput of the Open-FCoE is 156,529/129,951 IOPS for 4KB size random read/write requests, whereas FastFCoE is 286,500/285,446 IOPS, in 10Gbps link. Furthermore, 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 1122.1K/830K (for 4KB size reads/writes) IOPS to a remote target and achieve the near maximum throughput for 8KB or larger request sizes.

The remainder of this 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. In Section 3, we propose and present the details of the three optimization strategies within our prototype (FastFCoE). Section 4 evaluates the single core I/O performance and the I/O scalability of FastFCoE in a multi-core server. We discuss the related work in Section 5 and conclude our paper in Section 6.

2 REVISTING THE CURRENT FCoE I/O STACK

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. In addition, the block layer mainly implements I/O scheduling, which performs operations called merging and sorting to significantly improve the performance of system as a whole. The SCSI layer mainly constructs SCSI commands with I/O requests from

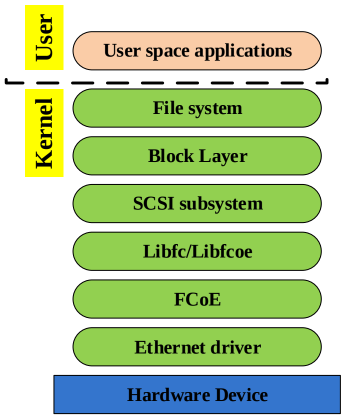


Fig. 1. Architecture of Linux Open-FCoE stack.

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]. In other words, the SCSI, FCP and FCoE layer mainly translate the I/O requests from BLOCK layer to FCoE command frames. The Ethernet driver transmits/receives FCoE frames to/from hardware. The main I/O performance factors in Open-FCoE stack can summarized as follows: (1) **I/O-issuing Side** translates the I/O requests into FCoE format frames; (2) **I/O Completion Side** informs the I/O-issuing threads of the I/O completions; (3) **Parallel Process and Synchronization** implements parallel access on multi-core servers. In this section, we describe and investigate the current Open-FCoE stack according to the above mentioned factors.

2.1 Issue 1: high synchronization overhead from single queue & shared lock mechanism

Figure 2 shows the I/O requests transmission process in the SCSI/FCP/FCoE layers of Open-FCoE stack when multiple cores/threads submit I/O requests to the remote target in multi-core systems. We describe it 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 when: (1) enqueueing the SCSI command into the shared queue in the SCSI layer; and (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 the SCSI layer and acquires the shared lock when 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 the *sk_buff* structure to the FCoE layer.
3) The FCoE layer encapsulates FC frame into FCoE frame, and then acquires the shared lock when: (1) enqueueing the FCoE frame; and (2) dequeueing the FCoE frame to transmit the frame to network with the standard interface *dev_queue_xmit()*.

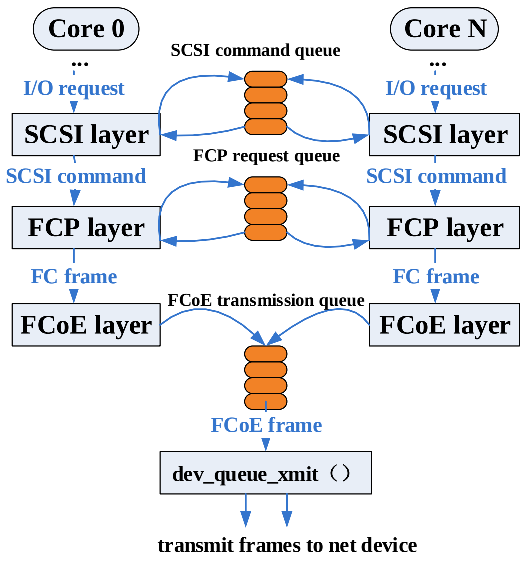


Fig. 2. Process of I/O requests transmission in the current Open-FCoE stack.

Obviously, the shared lock provides the synchronization operations on the shared queue in multi-core servers. However, such single queue & shared lock mechanism in SCSI/FCP/FCoE layer decreases the capacity of software scalability in multi-core systems.

For the purpose of improving scalability, 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 servers in our work. In such architecture, there are some representative features [11], [19], [20], [21], [22], [23], [24] 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 developing multi-threaded software in cc-NUMA multi-core systems [25].

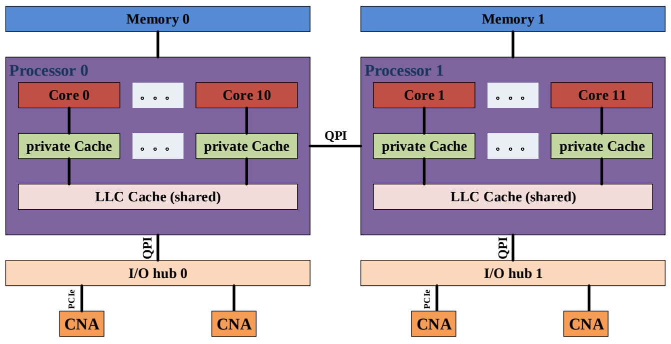


Fig. 3. Multi-core architecture with cache coherent Non-Uniform Memory Access (cc-NUMA).

We investigated 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 :

Single queue and global shared lock: As shown in Figure 2, in each of SCSI/FCP/FCoE layer, 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, the updating operation can lead to incoherence behavior that the cores are caching an old value. In the coherence protocol, the shared cache (LLC, Last Layer Cache) forwards the requests 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, and (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.

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 (such as Migratory sharing and remote memory accesses), 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 5, 4KB size 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 5(a) as a function of the number of cores that submit I/Os simultaneously. With Open-FCoE, we observe that the total throughput, as shown in Figure 5(b), does not increase too much with the increasing number of cores, since each I/O generates much more average LLC cache misses compared with only one core, as shown in Figure 5(a).

2.2 Issue 2: multi-layered software hierarchy to translate I/O requests to network frames

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 CPU overhead and latency for the remote target access in FCoE-based SAN storage.

As mentioned in Section 2, for each I/O operation the consumed time in the I/O issuing side mainly consists of three components, (1) I/O scheduling, (2) I/O translating and (3) frames transmitting. To observe the breakdown of software latency in the I/O issuing side within Open-FCoE stack, we measured the consumed time of each component when using a core to issue a single outstanding I/O request, as shown in Figure 7. We observe that in the Open-FCoE stack the I/O translating consumes a large fraction of execution time in the I/O issuing side. The execution times of the **I/O scheduling** : **I/O translating** : **frames transmitting** are $2\mu s : 5\mu s : 2\mu s$, respectively. That means that the implementing in SCSI/FCP/FCoE layers takes a long time to translate an I/O request into FCoE frame format. 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 is allocated and initialized with the SCSI command; then the FC format frame is allocated and initialized, such as copying the SCSI CDB to the frame. Extra costs are consumed in SCSI/FCP/FCoE layers, such as SCSI command, FCP internal 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 order to clearly describe this issue of multi-layered software hierarchy in the current Open-FCoE stack.

2.3 Issue 3: multiple context switchings in the I/O completion side

A context switch (also sometimes referred to as a process switch or a task switch) is the switching of the CPU from one task (a context or a thread) to another. When a new process has been selected to run, two basic jobs should be done [15] : (1) switching the virtual memory mapping from the previous processs to that of the new process. (2) switching the processor state from the previous processs to the current one. This involves saving and restoring stack information, the processor registers and any other architecture-specific state that must be managed and restored on a per-process basis.

Whenever a new context (interrupt or thread) is introduced in the I/O path, it can cause a polluted hardware cache and TLBs. And significant scheduling delays are added, particularly on a busy CPU [7], [8]. However, it is not trivial to remove these contexts in the I/O completion path since these contexts are employed to maintain system responsiveness and throughput. In this subsection, we investigate the path of I/O completion side and show two main types of latencies in the I/O completion path: task scheduling latency and the execution time for completion work in FCP/SCSI/BLOCK layer.

As we know, the block subsystem (block layer) schedules I/O requests by queueing them in a kernel I/O queue and placing the I/O-issuing thread in an I/O wait state. Upon finishing an I/O command (receiving a correct FCoE FCP_RSP packet [18]), there are at least three scheduling points to inform the I/O-issuing thread of I/O completion in Open-FCoE stack, as shown from Figure 8 (a) to (d). The current Open-FCoE stack is based on the standard network interface to receive/transmit FCoE packets from/to a network link. When receiving an FCoE frame, the adaptor generates a MSI-x interrupt to inform the core to call the interrupt service routines (ISRs) for implementing the **pre-processing work**, mainly including FCoE packets receiving and enqueueing as shown in Figure 8 (a). Then, the *fcoethread* thread (as shown in Figure 8 (b)) is waiting for being scheduled to do the **processing work** (mainly including de-queuing the received FCoE packets, FCoE FCP_RSP packet checking and FCP layer completion work) and raise the software interrupt (*BLOCK_SOFTIRQ [15]*). After that, the software interrupt (*BLOCK_SOFTIRQ*) handler (as shown in Figure 8 (c)) is scheduled to do the **post-processing work** (mainly including SCSI and BLOCK layer completion work) and try to wake up the I/O-issuing thread (waiting on this I/O completion, as shown in Figure 8 (d)). The I/O-issuing thread is later awakened to resume its execution.

TABLE 1
Major function of each context in the I/O completion side, in original Linux Open-FCoE.

Contexts	Major functions
MSI-x IRQ	FCoE packets receiving and enqueueing
fcoethread	dequeuing, FCoE FCP_RSP packet checking and FCP layer completion work
BLOCK_SOFTIRQ	SCSI and BLOCK layer completion work

TABLE 2
Total number of task switchings, task scheduling latencies, task running time and number of I/Os in the I/O completion side with our Open-FCoE

Task	Runtime (μs)	Switchings	Average delay(μs)	Total I/Os
fcoethread	1,371,556	50,283	5	49,472
I/O issuing	1,388,398	50,285	17	

To observe the breakdown of software latency in the I/O completion path within Open-FCoE stack, we measured the execution times of **pre-processing work** : **processing work** : **post-processing work** for each I/O completion when using a core to issue a single outstanding I/O request (4KB size read). The execution times of **pre-processing work** : **processing work** : **post-processing work** for each I/O completion are 4 us : 4 us : 7 us, respectively.

What's more, we recorded the total number of task switchings, average task scheduling latencies, task running time and the number of I/Os in the I/O completion path when using a core to issue a single outstanding I/O request (4KB size read), as shown in Table 2. For example, during 10 seconds, 49,472 read requests are implemented. The *fcoethread* spends 1,371,556 μs time to process the received FCoE frames. There are 50,283 context switchings for CPU (core) from other context to *fcoethread* context. The average scheduling delays for *fcoethread* and I/O issuing contexts are 5 μs and 17 μs, respectively. That means that in the I/O completion side there are average 22 μs time (not including the average scheduling delay of **BLOCK_SOFTIRQ** contex-t) consumed due to context scheduling.

3 I/O STACK OPTIMIZATION FOR ACCESSING THE FCOE SAN STORAGE

The analysis in Section 2 shows that the current I/O stack has two challenges: (1) How to decrease the processing 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 propose three optimization schemes within our prototype, which optimize the I/O performance with the following features : (1) significantly avoiding the synchronization overheads, (2) efficiently translating I/O requests into F-CoE frames, (3) substantially mitigating the I/O completion overhead in the I/O completion path.

First, we describe the architecture and the overall primary abstractions of our prototype (called FastFCoE, a FCoE protocol stack for accessing the FCoE based SAN storage), as shown in Figure 4. When we design the FastFCoE, one of our goals is to obtain the efficiency without the cost of decreasing compatibility and flexibility. (1) Our FastFCoE fully meets the related standards such as FC-BB-6 and FCP. (2) Our FastFCoE uses the standard software interfaces and needs not to revamp the upper and lower layer in software. (3) The salient feature of FastFCoE is simple to use and tightly integrated with existing Linux systems without the needs of specific devices or hardware features.

At the top of the architecture, there are multiple cores that implement the application threads and submit I/O requests to the block layer, which provides common services that valuable to applications and hides the complexity (and diversity) of storage devices. Our design is based on the multi-queue block layer [11] that allows each core to have a per-core queue for submitting I/O. Our proposed Fast-FCoE is under the multi-queue block layer and consists of three key components: FCoE Manager, Frame Encapsulation and Frame Process. The network link layer is under the FastFCoE. The frames from FastFCoE are transmitted to the network device (CNA, 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 for parallel transmitting/receiving, as shown in Figure 4. For instance, the Intel X520 10GbE converged network adaptor has 128 Tx queues and Rx queues.

3.1 Optimization 1 : using the private per-CPU structures & disabling kernel preemption method to avoid the high synchronization overheads

Through experiments and analysis in Subsection 2.1, we find the shared queue & lock mechanism in Open-FCoE would lead to the occurrence of LLC cache miss frequently and has a high synchronization overhead, which limits I/O throughput scalability in modern cc-NUMA multi-core systems.

To fully leverage parallel I/O capacity with multiple cores, we implement private per-CPU structures to process I/Os instead of the global shared variables accessing, such as single shared queue & lock mechanism. As shown in Figures 4 and 6, each core has its own private resources, such

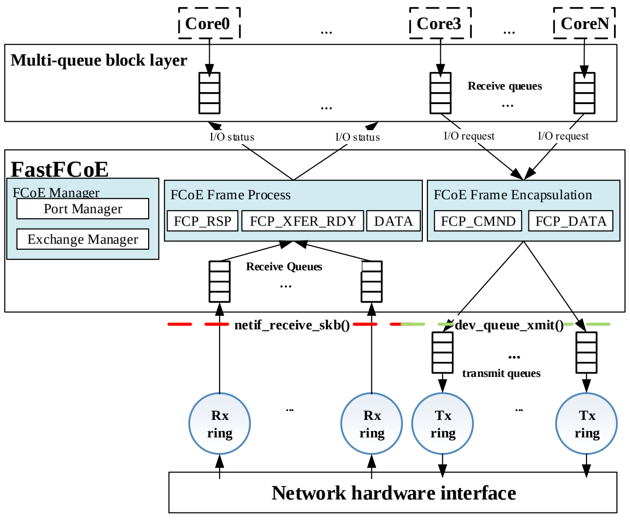


Fig. 4. FastFCoE architecture in multi-core server. The remote FCoE SAN storage target is mapped as a block device.

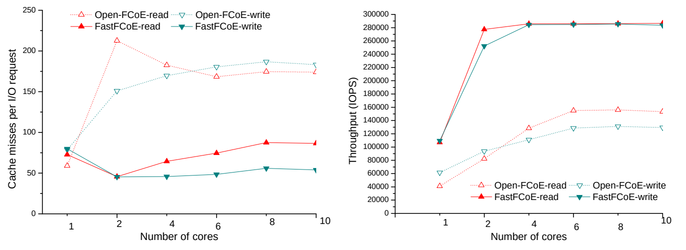
as queue*, Exchange Manager, CRC Manager, Rx/Tx ring, etc. We do not need to concern for the concurrent accessing from the threads running in other cores. For example, the Exchange Manager (as shown in Figure 6) uses private per-CPU variables to manage the Exchange ID† respectively for each I/O. During the ultra-short period of accessing the private per-CPU data, the kernel preemption is disabled and the current task (thread) will not be switched out. We also do not need to concern for the concurrent accessing from the threads running in the same core. This method avoids the synchronization overhead and significantly improves the parallel I/O capacity.

Disabling kernel preemption might cause a deferral of task scheduling and lengthen the latency in the current running thread. However, compared with the single queue & lock mechanism in existing Linux FCoE stack, there are several benefits to use per-CPU data. First, our scheme removes the locking requirement for accessing the shared queue. Second, per-CPU data is private for each core, which greatly reduces the cache invalidation (detailed in Subsection 2.1 Migratory sharing). Moreover, our FastFCoE is designed for Linux operating system, which is not a hard real-time operating system and makes no guarantees on the capability to schedule real-time tasks [15]. Each core has its own private per-CPU structures, 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. In fact, the per-CPU structure & disabling preemption is commonly used in the Linux kernel 2.6 or newer versions.

As shown in Figure 5, 4KB size random I/Os were submitted to a remote target, to compare our method (FastFCoE) with Open-FCoE. The average number of cache misses per I/O and the total throughput are depicted in Figure 5(a) and Figure 5(b), respectively, as a function of the number

*. Multi-queue block [11] layer allows each core to have a per-core queue for submitting I/O.

†. A unique identifier in Fibre Channel Protocol-SCSI (FCP) [18] for each I/O request.



(a) Average LLC cache misses per I/O

(b) Total throughput

Fig. 5. **Average LLC cache misses per I/O and throughput (IOPS) comparison between original Linux Open-FCoE and our FastFCoE.** 4KB size random I/Os are submitted as a function of number of cores issuing I/Os in 10Gbps link. The cores are distributed uniformly in a 2-socket system.

of cores that submit I/Os simultaneously. As shown in Figure 5(b), we observe that the throughput with Open-FCoE does not increase too much with the increasing number of cores, whereas our method (FastFCoE) has a significant improvement (achieves the near maximum throughput in 10Gbps link). Our method (FastFCoE) generates much less average LLC cache misses per I/O, compared with Open-FCoE, as shown in Figure 5(a).

3.2 Optimization 2 : directly mapping I/O requests into FCoE frames

As mentioned in Subsection 2.2, due to the layered software architecture in the current Open-FCoE stack, the extra inter-layer and intra-layer cost are consumed to translate I/O requests to FCoE frames.

Instead of SCSI/FCP/FCoE layers in the current Open-FCoE stack, we directly initializes the FCoE frame with the I/O request from the block layer. Figure 6 shows the mapping from I/O request to network messages. 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 [26], directly transfers the multiple non-liner memory segments to the hardware (CNA) by DMA. In addition, FCoE segmentation offload (FSO) [16] is a technique for reducing CPU overhead and increasing the outbound throughput of high bandwidth converged network adaptor (CNA) by allowing the hardware (CNA) to split a large frame 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 the large transferred data. These non-linear buffers include request segments in memory pages and the CRC, EOF (not shown in Figure 6) fields in FCP_DATA frame. What's more, 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, FastFCoE also supports Direct Data Placement Offload (DDP) [16], which saves CPU overhead by allowing the CNA to transfer the FCP_DATA frame payload (the transferred data) to the request memory segments.

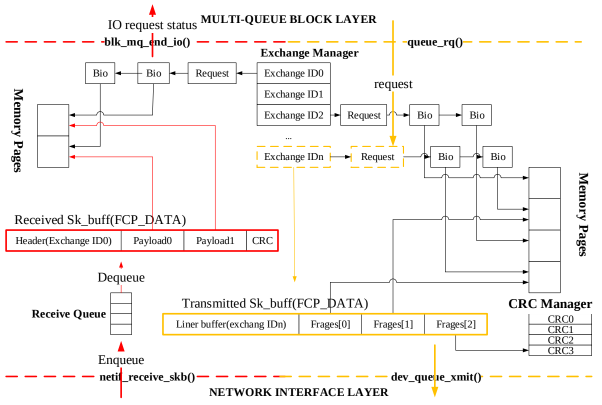


Fig. 6. Mapping from I/O requests to network messages in FastFCoE.

This optimization scheme (directly mapping the requests from the block-layer to the FCoE frames) cuts the extra inter-layer and intra-layer cost, and significantly reduces the software latency in the I/O issuing side. Figure 7 presents the software latency comparison between Open-FCoE and our scheme (FastFCoE) in the I/O issuing side, when using a core to issue a single outstanding I/O request. Our FastFCoE is effective in reducing the software latency in the I/O issuing side (reduction to 66.67%). The source of the improvement is from the high-efficiency of I/O translating. With our scheme, $2\mu s$ time is consumed in the I/O translating and $3\mu s$ is saved, as shown in Figure 7.

3.3 Optimization 3 : eliminating the I/O completion side latency

As mentioned in Subsection 2.3, there are two main types of latencies in the I/O completion side, task scheduling

‡. Representing the data delivery request.

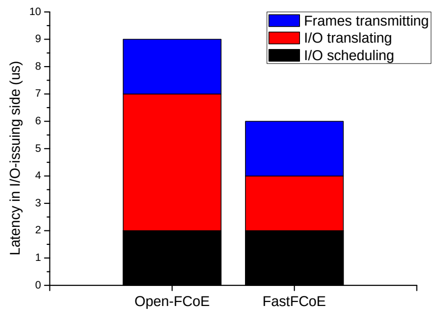


Fig. 7. Software overhead comparison on I/O-issuing side between original Linux Open-FCoE and our FastFCoE. For each I/O operation the consumed time in the I/O issuing side consists of three components, (1) I/O scheduling, (2) I/O translating and (3) frames transmitting. Setup: direct I/O, noop I/O scheduler, 512 Byte size random read, iodepth=1.

latency and the execution time for completion work in FCP/SCSI/BLOCK layer. Our goal is not only to eliminate the number of context switchings in the I/O completion path, but also to reduce the total execution time in these contexts. In this subsection, we briefly introduce the idea.

For direct-attached SCSI drive (based on SCSI layer) devices, the software interrupt (*BLOCK_SOFTIRQ*) context is necessary to do the deferred work (SCSI layer and BLOCK layer completion work), which avoids system lockdown caused by heavy ISRs [8]. But, network adaptors use the NAPI mechanism [26] to avoid the high overhead of ISRs. We point out that for network adaptors the *BLOCK_SOFTIRQ* context is redundant due to the *fcoethread* context, which can directly do the **post-processing** work. So we remove the *BLOCK_SOFTIRQ* context in the I/O completion path. The **post-processing** work is directly done by *fcoethread* context (in FastFCoE we name it as *fastfcoethread*, as shown in Figure 8). Furthermore, in the consideration of Optimization 2 (the SCSI/FCP/FCoE layers are replaced by one layer), the execution time of each I/O completion is reduced significantly due to the deletion of the extra completion work, such as SCSI and FCP layer completion work. In FastFCoE the total execution time of **processing work** + **post-processing work** for each I/O completion is 5 μs, whereas in Open-FCoE the total execution time of **processing work** + **post-processing work** for each I/O completion is 11 μs, when using a core to issue a single outstanding I/O request (4KB size read).

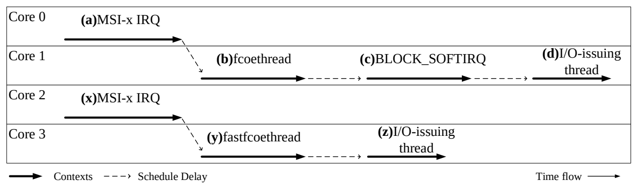


Fig. 8. I/O completion scheme comparison of original Linux Open-FCoE ((a) to (d)) and our FastFCoE ((x) to (z)). The major function of each context in the I/O completion path is listed in Tables 1 and 3, in original Linux Open-FCoE and our FastFCoE, respectively.

TABLE 3
Major function of each context in the I/O completion side, in our FastFCoE.

Contexts	Major functions
MSI-x IRQ	FCoE packets receiving and enqueueing
fastfcoethread	dequeuing, FCoE FCP_RSP packet checking, FCoE layer and BLOCK layer completion work

This method not only reduces the total execution time of **processing** and **post-processing** work, but also removes the extra context switching to avoid the extra context scheduling delays. The total number of task switchings, average task scheduling latencies, task running time and number of I/Os in the I/O completion path were also recorded when using a core to issue a single outstanding I/O request (4KB size read), as shown in Table 4. During 10 seconds, our method (FastFCoE) spends 736,152 μs time to do all the I/O completion works for 53,004 read requests (as shown in Table 4), whereas Open-FCoE spends 1,371,556 μs time to do the partial I/O completion works for 49,472 read requests (as shown in Table 2). However, the average scheduling delays with FastFCoE are 4 μs and 6 μs for *fcoethread* and I/O issuing contexts respectively, whereas with Open-FCoE are 5 μs and 17 μs (as shown in Table 2). The major source of the results is due to the fact that there is only one context (*fastfcoethread*) to implement the fewer completion works in our FastFCoE stack rather than the two contexts (*fcoethread* and *BLOCK_SOFTIRQ*) in Open-FCoE stack.

TABLE 4
Total number of task switchings, scheduling latencies, running time and number of I/Os in the I/O completion side with our FastFCoE

Task	**Runtime** (μs)	**Switchings**	**Average delay** (μs)	**Total I/Os**
fastfcoethread	736,152	53,890	4	53,004
I/O issuing	1,608,798	53,891	6	

4 Experimental Evaluation

In modern data centers, there are two common deployment solutions for servers, including traditional non-virtualized servers (physical machines) and virtualized servers (virtual machines). In this section, we performed several experiments to test the overall performance of our prototype system (FastFCoE). The experimental results [§] 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 under the different configurations of Process Affinity and IRQ Affinity [32], [33], which are related to I/O performance? (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.

4.1 Experimental Method and Setup

To understand the overall performance of our FastFCoE, we evaluated the main features with two micro-benchmark FIO [27] and Orion [28]. 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 analyzed the impact of throughput performance under different degrees of CPU loads with real world TPC-C [29] and TPC-E [30] benchmark traces.

We performed the Open-FCoE stack in the Linux kernel as baseline to carry out the comparisons. Our experimental platform consisted of two systems (initiator and target), connected back-to-back with multiple CNAs. Both initiator server and target server were configured with Dell PowerEdge R720, Dual Intel Xeon Processor E5-2630 (6 cores, 15MB Cache, 2.30GHz, 7.20GT/s Intel QPI), 128GB DDR3, Intel X520 10Gbps CNAs, with hyperthreading capabilities enabled. The Open-FCoE or FastFCoE stack ran in the host or virtual machines with CentOS 7 (3.13.9 kernel). The target system was based on the modified Linux I/O target (LIO) 4.0 with CentOS 7 (3.14.0 kernel) and used 40GB RAM as a disk. Note that we used 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) [16], the hardware offload functions for FCoE protocol, was enabled when the request size was equal to or larger than 4KB.

4.2 Performance Results

First, we performed FIO tool to compare the single core performance of FastFCoE with Open-FCoE in terms of the average throughput, CPU overhead and latency by sending a single outstanding 512B I/O with a single core. Then, we evaluated 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 used two benchmark traces (TPC-C [30] and TPC-E [31]) to evaluate throughput performance between FastFCoE and the Open-FCoE under different degrees of CPU loads.

4.2.1 Single core Evaluation

In this subsection, we modify the tunning parameters for Process Affinity[¶] and IRQ Affinity[‖] [26] to evaluate the I/O performance of a single core under the six typical configurations, as shown in Figure 9. For example, the configuration of Figure 9 (a) means: The application runs and submits I/O requests in core 0, on NUMA node 0. The MSI-x interrupt [16] is handled by core 2[∗∗], on NUMA node 0. The converged network adaptor (CNA) is on the other NUMA node, NUMA node 1.

The throughput, CPU usage and latency are measured by issuing a single outstanding 512B I/O with a single core in the non-virtualized and virtualized systems with 10Gbps CNA, respectively. As shown in Figure 10, our FastFCoE has a significant improvement of throughput performance than Open-FCoE for all of the six configurations (as shown in Figure 9). In addition, for both of Open-FCoE and FastFCoE, we observe that throughput performance is better when the core submitting I/Os is on the same NUMA node with the adapter (CNA) (the configuration c, d and f, as shown in Figure 10) than others (the configuration a, b and e, as shown in Figure 10).

Rather than the layered architecture in Open-FCoE, which results in the extra inter-operations and intra-operations to translate the I/O requests to FCoE format frames, FastFCoE directly maps the requests from the block-layer to the FCoE frames. What is more, FastFCoE uses a new I/O completion scheme, which avoids the extra context switching (*BLOCK_SOFTIRQ* context) overhead and reduces the execution overhead (due to the deletion of the extra completion work). As a result, FastFCoE has less CPU overhead for each I/O request than Open-FCoE. Figure 11

§. In this section, each experiment runs 10 times. The best and worst results are discarded to remove outliers. The remaining 8 results are used to calculate the standard deviation and average values.

¶. Processor affinity, or CPU pinning enables the binding and unbinding of a process or a thread to a central processing unit (CPU) or a range of CPUs, so that the process or thread will execute only on the designated CPU or CPUs rather than any CPU.

‖. IRQs have an associated "affinity" property, *smp_affinity*, which defines the CPU cores that are allowed to execute the ISR for that IRQ.

∗∗. When receiving an FCoE frame, the adaptor generates a MSI-x interrupt to inform the core 2 to receive the FCoE frame.

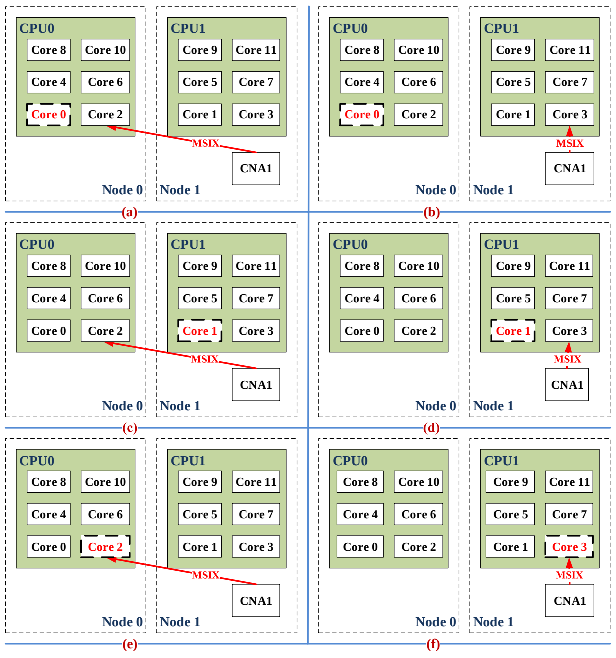


Fig. 9. Six typical configurations for Process Affinity and IRQ Affinity [26] in our prototype (Dual Intel Xeon Processor E5-2630). For example, the configuration (a) means: The application runs and submits I/O requests in core 0, on NUMA node 0. The MSI-x interrupt [16] is handled by core 2, on NUMA node 0. The converged network adaptor (CNA) is on the other NUMA node, NUMA node 1.

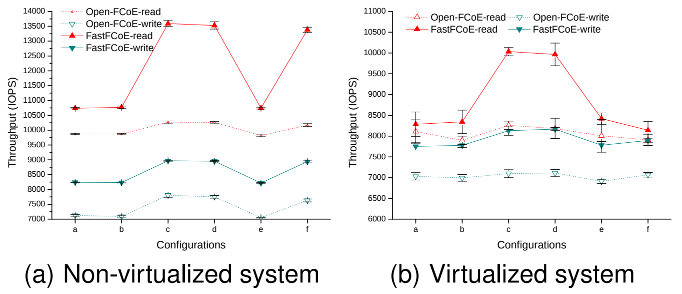


Fig. 10. Throughput is measured by issuing a single outstanding 512B I/O with a single core under the six configurations, as shown in Figure 9.

shows the average CPU utilization for Open-FCoE and FastFCoE, with the six configurations in the non-virtualized and virtualized systems. For the non-virtualized system, the average CPU utilization of FastFCoE has a decrease of 3.15%∼6.74% and 6.05%∼8.34% for read and write, respectively. For the virtualized system, the average CPU utilization of FastFCoE has a decrease of 2.52%∼4.47% and 2.26%∼2.28% for read and write, respectively. The hareware capacity of DDP [16] is disabled in 512B read operation, thus requiring higher CPU overhead than write operation.

The latency is measured as the time from the application, through the kernel, into the network. Our FastFCoE has a short I/O path both on the I/O issuing side and I/O completion side. Hence, FastFCoE has a smaller average latency than Open-FCoE. Figure 12 shows the average latency for Open-FCoE and FastFCoE, with the six configurations in the non-virtualized and virtualized systems. For the non-virtualzied system, the average latency of FastFCoE has a decrease of 7.81∼22.78 and 16.38∼18.84 microseconds for read and write, respectively. For the virtualized system, the average latency of FastFCoE has a decrease of 2.55∼20.88

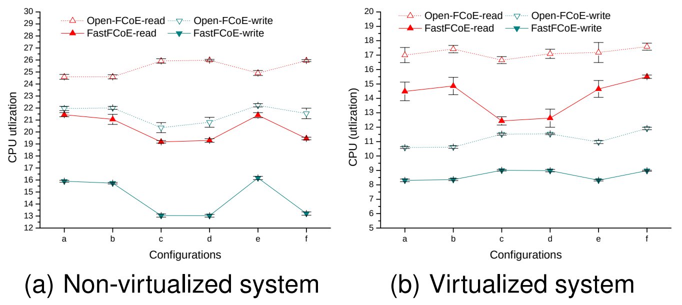


Fig. 11. CPU utilization is measured by issuing a single outstanding 512B I/O with a single core under the six configurations as shown in Figure 9.

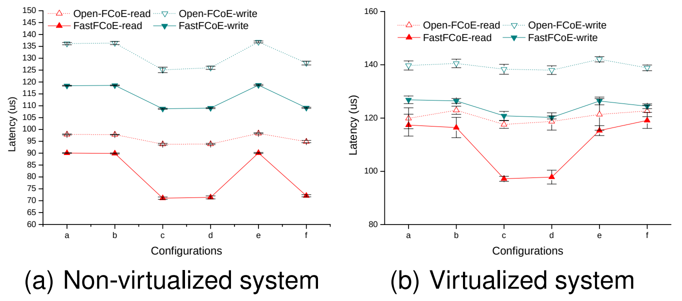


Fig. 12. Average I/O latency is measured by issuing a single outstanding 512B I/O with a single core under the six configurations as shown in Figure 9.

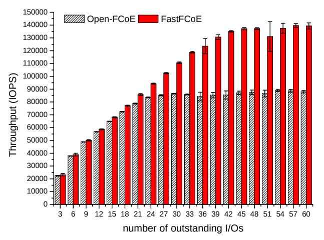
and 12.88∼17.75 microseconds for read and write, respectively. The write operation causes higher complexity in FCP protocol [18] than read operation. Therefore, the write operation has a larger latency than read operation.

4.2.2 *I/O scalability Evaluation*

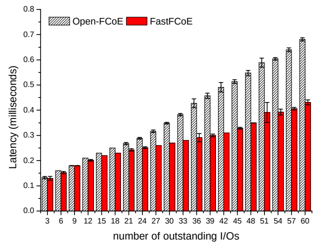
The improvements of the I/O scalability with the increasing number of concurrent I/Os and the increasing number of cores submitting I/Os are important to I/O subsystem. In this subsection, we performed FIO and Orion [28] to evaluate the I/O scalability of FastFCoE in the non-virtualized and virtualized systems, respectively.

We performed 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).

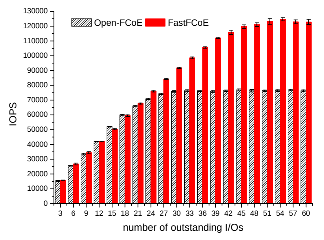
We evaluated the performance in OLTP (as shown in Figure 13) and DSS (as shown in Figure 14) application scenarios with 50% write requests on FastFCoE and Open-FCoE in 10Gbps Ethernet link, respectively. With the increasing number of concurrent I/Os, the I/Os become more intensive. Since FastFCoE has a 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) of FastFCoE outperforms Open-



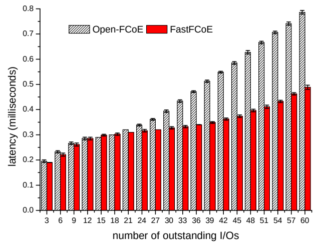
(a) IOPS for OLTP in the non-virtualized system



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



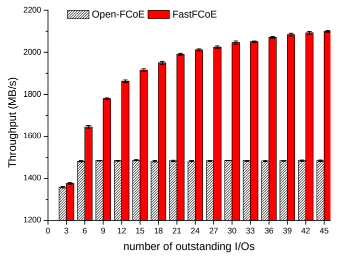
(c) IOPS for OLTP in the virtualized system



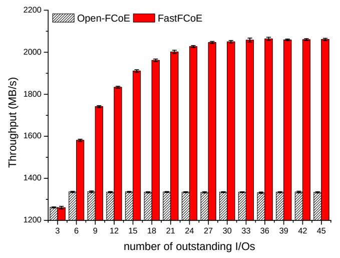
(d) Latency for OLTP in the virtualized system

Fig. 13. I/O scalability Evaluation with Orion (50% write). The figures show 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.

FCoE by 1.58X and 1.63X at most, in the non-virtualied and virtualied system, respectively. At the same time the average latencies have 37.0% and 36.1% reduction, respectively. For DSS, the throughput of FastFCoE outperforms Open-FCoE by 1.63X and 1.55X at most, in the non-virtualied and the virtualied system, respectively. The reason of the results is that FastFCoE has smaller process overheads than Open-FCoE.



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

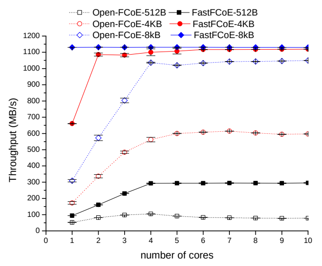


(b) Throughput for DSS in the virtualized system

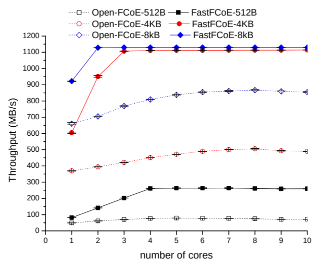
Fig. 14. I/O scalability Evaluation with Orion (50% write). The figures show the average throughput obtained by FastFCoE and Open-FCoE in different numbers of outstanding IOs for DSS test, with the non-virtualized and virtualized systems, respectively.

One challenge for storage 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 performed 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 bound one thread for each permitted core.

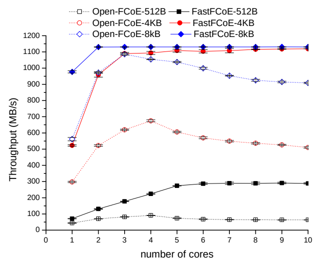
Figure 15 shows the total throughput by submitting 64 outstanding asynchronous random 512B, 4KB and 8KB size requests, respectively, with different numbers of cores, with 10Gbps CNA. For the non-virtualized system, when



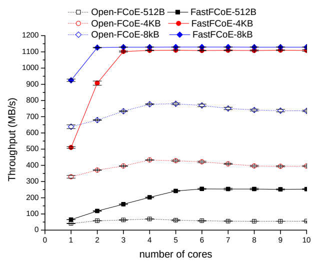
(a) read in the non-virtualized system



(b) write in the non-virtualized system



(c) read in the virtualized system



(d) write in the virtualized system

Fig. 15. Scalability Evaluation with FIO (random workload). The figures show the total throughput of FastFCoE and Open-FCoE when changing the number of cores submitting 64 outstanding 512B,4KB and 8KB I/O requests in the non-virtualized and virtualized systems with 10Gbps CNA.

using one core, our FastFCoE shows higher throughput than Open-FCoE by 1.79/1.67X, 3.84/1.63X and 3.66/1.40X on 512B, 4KB and 8KB read/write requests, respectively. For 512B read requests, FastFCoE achieves almost the highest throughput of a single CNA, 616,221 IOPS (300.89MB/s), whereas the Open-FCoE is 215,117 (105.04MB/s). This shows that Open-FCoE has a limited throughput (IOPS), no more than 22K. 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 1.63/1.58X, 1.76/1.55X and 1.74/1.45X on 512B, 4KB, 8KB read/write requests, respectively. For 512B read/write requests, FastFCoE achieves 617,724/540,900 IOPS (301.62/264.11MB/s) at most, whereas the Open-FCoE is 189,444/145,331 IOPS (93.08/70.96MB/s). Our FastFCoE uses the private per-CPU structures & disabling kernel preemption to avoid synchronization overhead. This approach significantly improves the I/O scalability with the increasing number of cores.

To further study the I/O scalability of FastFCoE, while avoiding the influence from limited capacity of adapter (CNA), we bonded 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 16. For 4KB read requests, the IOPS of FastFCoE can improve with the increasing number of cores to submit requests until around 1.1221M IOPS (4,383.3MB/s). Although the write operation has higher complexity in FCP protocol [18] than read operation, for 4KB random write, FastFCoE still achieves up to 830,210 IOPS (3,243MB/s).

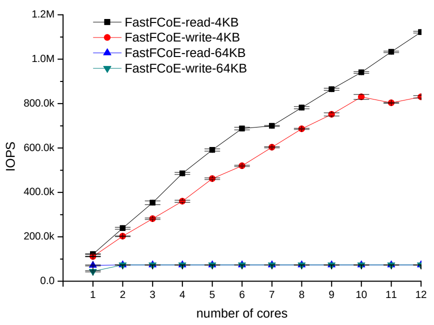


Fig. 16. Scalability Evaluation with FIO in 40Gbps link. IOPS obtained by FastFCoE depends on the number of cores with 4KB and 64KB random read/write requests when bonding four 10Gbps CNAs as one 40Gbps CNA in the non-virtualization system.

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 4,454.9MB/s for 64KB random read requests. FastFCoE hence has sufficient capacity to fit with 40Gbps link in the FCoE-based SAN storage.

4.2.3 TPC-C and TPC-E tests Using OLTP Disk Traces

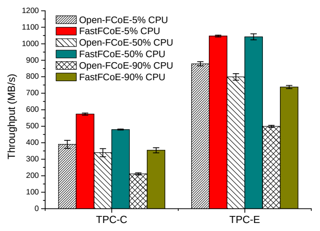
Many applications consume a large amount of CPU resource and affect I/O subsystem. To show the throughput of FastFCoE over Open-FCoE under different degrees of CPU loads, we analyzed the throughput in both non-virtualized and virtualized systems with a 10Gbps CNA by using OLTP benchmark traces: TPC-C [30] and TPC-E [31]. These traces are obtained from test using HammerDB [32] 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 [27]. We perform 5%, 50% and 90% CPU loads, respectively, to represent the 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 17 shows the superiority of FastFCoE over Open-FCoE in both 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 1.47X, 1.41X, 1.68X and 1.55X, 1.56X, 1.13X in the non-virtualized and virtualized systems with 5%, 50% and 90% CPU loads, respectively. For TPC-E benchmark, FastFCoE outperforms Open-FCoE by 1.19X, 1.30X, 1.48X and 1.42X, 1.46X, 1.43X in the non-virtualized and virtualized systems with 5%, 50% and 90% CPU loads, respectively.

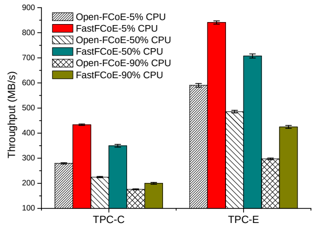
5 Related Work

This work touches on the software and hardware interfaces of network and storage on multi-core systems. Below we describe the related work.

OS bypass scheme. 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



(a) Throughput in the non-virtualized system



(b) Throughput in the virtualized system

Fig. 17. Throughput Evaluation with TPC-C and TPC-E. The figures show the throughputs achieved by FastFCoE and Open-FCoE, with 5%, 50% and 90% CPU loads, in the non-virtualized and virtualized systems, respectively.

cases. 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. Caulfield *et al.* [6] propose to bypass the block layer and implement their own driver and single queue mechanism to improve I/O performance.

Our optimization scheme is under the block layer and calls the standard network interfaces to transmit/receive network packets. Therefore, it can support all upper software components (such as existing file systems and applications) and be deployed with existing infrastructures (adaptors, switches and storage devices), without the costs of extra hardware.

Scalability on Multi-core Systems. Over the last few years, a number of studies have attempted to improve the scalability of operating systems in the 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 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 I/O scalability in multi-core servers.

High speed I/O software. Software overhead from high-speed I/O, such as network adaptor and Non-Volatile Memory (NVM) storage device, obtains a lot of attentions, which consumes substantial system resources and influences on the system performance [26].

Rizzo and Luigi [34], [35] propose *netmap*, a framework that shows user-pace applications to exchange raw packets with the network adapter (maps packet buffers into the process memory space), thus making a single core running at 900 MHz to send or receive 14.88Mpps (the peak packet rate on in 10Gbps links). The Intel Data Plane Development Kit [36] (DPDK) is an open source, optimized software library for Linux User Space applications. Due to lots of optimization strategies (such as using a polled-mode drive to avoid the high overhead from interrupt-driven driver, processing a bunch of packets to amortize the access cost over multiple packets, and using Huge Pages to make best use of limited number of TLB resources), this library can improve packet processing performance by up to ten times and achieve over 80 Mpps throughput on a single Intel Xeon processor (double that with a dual-processor configuration). Both *netmap* and intel DPDK are used by user-space applications for fast processing of raw packets (ethernet frames), whereas our optimization strategies within FastFCoE use the standard network interface in kernel to do the FCoE protocol packets processing.

Jisoo Yang *et al.* [7] show that when using NVM device polling for the I/O completion delivers higher performance than traditional interrupt-driven I/O. Woong Shin *et al.* [8] present a low latency I/O completion scheme for fast storage to support current flash SSDs. Our optimization strategies focuses on the issues at the software interface between the host and the CNA, which emerges as a bottleneck in high-performance FCoE based SAN storage.

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. This multi-queue block layer leverages the performance offered by SSDs and NVM Express, by allowing much higher I/O submission rates. In this paper, we introduces the multi-queue block layer to FCoE protocol process and decrease the I/O path by (1) directly mapping the requests from the block-layer to the FCoE frames and (2) a new I/O completion scheme, which eliminates the number of contexts and the total execution time in the completion side.

6 Conclusion

In the context of high-speed network and fast storage technologies, there is a need for a high-performance storage stack. In this paper, we expose the inefficiencies of the current Open-FCoE stack from three factors (synchronization overhead, processing overhead on the I/O-issuing side and I/O completion side), which lead to a high I/O overhead and limited I/O scalability in FCoE-based SAN storage. We propose a synergetic and efficient solution for accessing the FCoE based SAN storage on multi-core servers. Compared with the current Open-FCoE stack, our solution has following advantages : (1) better performance of parallel I/O in multi-core servers; (2) lower I/O processing overhead both on the I/O-issuing side and I/O completion side. Experimental results demonstrate that our solution achieves an efficient and scalable I/O throughput on multi-core servers.

Acknowledgment

This work was supported by the 863 Project No.2015AA015301; the National Key Research and Development Program of China under Grant 2016YFB1000202; the 863 Project No.2015AA016701; NSFC No.61502191, No.61502190, No.61472153, No.61402189; State Key Laboratory of Computer Architecture, No.CARCH201505; Wuhan Applied Basic Research Project No.2015010101010004; and Hubei Provincial NSFC No.2016CFB226. This is an extended version of our manuscript published in the Proceedings of the 44th International Conference on Parallel Processing (ICPP), 2015. Fang Wang is the corresponding author.

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. 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.

[23] 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.

[24] 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.

[25] Intel Guide for Developing Multithreaded Applications. https://software.intel.com/en-us/articles/intel-guide-for-developing-multithreaded-applications.

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

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

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

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

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

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

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

[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.

[36] DATA PLANE DEVELOPMENT KIT. http://www.dpdk.org/.

Yunxiang Wu received the BE degree in computer science and technology from the Wuhan University of Science and Technology (WUST), China, in 2009. He is currently a PhD student majoring in computer architecture in Huazhong University of Science and Technology, Wuhan, China. His current research interests include computer architecture and storage systems.

Fang Wang received her B.E. degree and Master degree in computer science in 1994, 1997, and Ph.D. degree in computer architecture in 2001 from Huazhong University of Science and Technology (HUST), China. She is a professor of computer science and engineering at HUST. Her interests include distribute file systems, parallel I/O storage systems and graph processing systems. She has more than 50 publications in major journals and international conferences, including FGCS, ACM TACO, SCIENCE CHINA Information Sciences, Chinese Journal of Computers and HiPC, ICDCS, HPDC, ICPP.

Yu Hua received the BE and PhD degrees in computer science from the Wuhan University, China, in 2001 and 2005, respectively. He is a full professor at the Huazhong University of Science and Technology, China. His research interests include computer architecture, cloud computing, and network storage. He has more than 100 papers to his credit in major journals and international conferences including IEEE Transactions on Computers (TC), IEEE Transactions on Parallel and Distributed Systems (TPDS), USENIX ATC, USENIX FAST, INFOCOM, SC and ICDCS. He has been on the program committees of multiple international conferences, including USENIX ATC, RTSS, INFOCOM, ICDCS, MSST, ICNP and IPDPS. He is a senior member of the IEEE, ACM and CCF, and a member of USENIX.

Dan Feng received the BE, ME, and PhD degrees in Computer Science and Technology in 1991, 1994, and 1997, respectively, from Huazhong University of Science and Technology (HUST), China. She is a professor and vice dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 100 publications in major journals and international conferences, including IEEE-TC, IEEE-TPDS, ACM-TOS, JCST, FAST, USENIX ATC, ICDCS, HPDC, SC, ICS, IPDPS, and ICPP. She serves on the program committees of multiple international conferences, including SC 2011, 2013 and MSST 2012. She is a member of IEEE and a member of ACM.

Yuchong Hu received the B.Eng. degree in Computer Science and Technology from Special Class for the Gifted Young (SCGY), the University of Science and Technology of China in 2005, and the Ph.D. degree in Computer Software and Theory from the University of Science and Technology of China in 2010. He is an associate professor of the School of Computer Science and Technology at the Huazhong University of Science and Technology. His research interests include network coding/erasure coding, cloud computing and network storage. He has more than 20 publications in major journals and conferences, including IEEE Transactions on Computers (TC), IEEE Transactions on Parallel and Distributed Systems (TPDS), IEEE Transactions on Information Theory, FAST, INFOCOM, MSST, ICC, DSN, and ISIT.

Wei Tong received the BE, ME, and PhD degrees in computer science and technology from the Huazhong University of Science and Technology (HUST), China, in 1999, 2002, and 2011, respectively. She is a lecturer of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, network storage system, and solid state storage system. She has more than 10 publications in journals and international conferences including ACM TACO, MSST, NAS, FGCN.

Jingning Liu received the BE degree in computer science and technology from the Huazhong University of Science and Technology (HUST), China, in 1982. She is a professor in the HUST and engaged in researching and teaching of computer system architecture. Her research interests include computer storage network system, high-speed interface and channel technology, embedded system.

Dan He is currently a PhD student majoring in computer architecture in Huazhong University of Science and Technology. Wuhan, China. His current research interests include Solid State Disks, PCM, and file system. He publishes several papers including TACO, HiPC, ICA3PP, etc.