QZFS: QAT Accelerated Compression in File System for Application Agnostic and Cost Efficient Data Storage

Xiaokang Hu and Fuzong Wang, *Shanghai Jiao Tong University, Intel Asia-Pacific R&D Ltd.;* Weigang Li, *Intel Asia-Pacific R&D Ltd.;* Jian Li and Haibing Guan, *Shanghai Jiao Tong University*

https://www.usenix.org/conference/atc19/presentation/wang-fuzong

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

QZFS: QAT Accelerated Compression in File System for Application Agnostic and Cost Efficient Data Storage

Xiaokang Hu
Shanghai Jiao Tong University
Intel Asia-Pacific R&D Ltd.

Fuzong Wang*
Shanghai Jiao Tong University
Intel Asia-Pacific R&D Ltd.

Weigang Li
Intel Asia-Pacific R&D Ltd.

Jian Li
Shanghai Jiao Tong University

Haibing Guan
Shanghai Jiao Tong University

Abstract

Data compression can not only provide space efficiency with lower Total Cost of Ownership (TCO) but also enhance I/O performance because of the reduced read/write operations. However, lossless compression algorithms with high compression ratio (e.g. gzip) inevitably incur high CPU resource consumption. Prior studies mainly leveraged general-purpose hardware accelerators such as GPU and FPGA to offload costly (de)compression operations for application workloads. This paper investigates ASIC-accelerated compression in file system to transparently benefit all applications running on it and provide high-performance and cost-efficient data storage. Based on Intel® QAT ASIC, we propose QZFS that integrates QAT into ZFS file system to achieve efficient gzip (de)compression offloading at the file system layer. A compression service engine is introduced in QZFS to serve as an algorithm selector and implement compressibility-dependent offloading and selective offloading by source data size. More importantly, a QAT offloading module is designed to leverage the vectored I/O model to reconstruct data blocks, making them able to be used by QAT hardware without incurring extra memory copy. The comprehensive evaluation validates that QZFS can achieve up to 5x write throughput improvement for FIO micro-benchmark and more than 6x cost-efficiency enhancement for genomic data post-processing over the software-implemented alternative.

1 Introduction

Data compression has reached proliferation in systems involving storage, high-performance computing (HPC) or big data analysis, such as EMC CLARiiON [14], IBM zEDC [7] and RedHat VDO [18]. A significant benefit of data compression is the reduced storage space requirement for data volumes, along with the less power consumption for cooling per unit of logical storage [12, 51]. Furthermore, if the input data to Hadoop [3], Spark [4] or stream processing job [40] is compressed, the data processing performance can be effectively enhanced as the compression not only saves bandwidth but also decreases the number of read/write operations from/to storage systems.

It is widely recognized that the benefits of data compression come at the expense of computational cost [1, 9], especially for lossless compression algorithms with high compression ratio [41]. In a number of fields (e.g., scientific big data or satellite data), lossless compression is the preferred choice due to the requirement for data precision and information availability [12, 43]. Prior studies mainly leveraged general-purpose hardware accelerators such as GPU and FPGA to alleviate the computational cost incurred by (de)compression operations [15, 38, 41, 45, 52]. For example, Ozsoy *et al.* [38] presented a pipelined parallel LZSS compression algorithm for GUGPU and Fowers *et al.* [15] detailed a scalable fully pipelined FPGA accelerator that performs LZ77 compression. Recently, the emerging AISC (Application Specific Integrated Circuit) compression accelerators, such as Intel® QuickAssist Technology (QAT) [24], Cavium NITROX [34] and AHA378 [2], are attracting attentions because of their advantages on performance and energy-efficiency [32].

Data compression can be integrated into different system layers, including the application layer (most common), the file system layer (e.g., ZFS [48] and BTRFS [42]) and the block layer (e.g., ZBD [27] and RedHat VDO [18]). Professional storage products such as IBM Storwize V7000 [46] and HPE 3PAR StoreServ [19] may contain competitive compression feature as well. If compression is performed at the file system or lower layer, all applications, especially big data processing workloads, running in the system can transparently benefit from the enhanced storage efficiency and reduced storage I/O cost per data unit. This feature also implies that only lossless compression is acceptable to avoid influences on applications. To the best of our knowledge, there is no practical solution at present that provides hardware-accelerated data compression at the layer of local or distributed file systems.

In this paper, we propose QZFS (QAT accelerated ZFS) that

*Co-equal First Author

integrates Intel® QAT accelerator into the ZFS file system to achieve efficient data compression offloading at the file system layer so as to provide application-agnostic and cost-efficient data storage. QAT [24] is a modern ASIC-based acceleration solution for both cryptography and compression. ZFS [6,48] is an advanced file system that combines the roles of file system and volume manager and provides a number of features, such as data integrity, RAID-Z, copy-on-write, encryption and compression.

In consideration of the goal of cost-efficiency, QZFS selects to offload the costly gzip [1] algorithm to achieve high space efficiency (i.e., high compression ratio) and low CPU resource consumption at the same time. QZFS disassembles the (de)compression procedures of ZFS to add two new modules for integrating QAT acceleration. First, a compression service engine module is introduced to serve as a selector of diverse compression algorithms, including QAT-accelerated gzip and a number of software-implemented compression algorithms. It implements compressibility-dependent offloading (i.e., compression/non-compression switch) and selective offloading by source data size (i.e., hardware/software switch) to optimize system performance. Second, a QAT offloading module is designed to efficiently offload compression and decompression operations to the QAT accelerator. It leverages the vectored I/O model, along with address translation and memory mapping, to reconstruct data blocks prepared by ZFS, making them able to be accessed by QAT hardware through DMA operations. This kind of data reconstruction avoids expensive memory copy to achieve efficient offloading. Besides, considering QAT characteristics, this module further provides buffer overflow avoidance, load balancing and fail recovery.

实验测试集：
1.FIO micro
2.genomic data

In the evaluation, we deploy QZFS as the back-end file system of Lustre [44] in clusters with varying nodes, and measure the performance with FIO micro-benchmark and practical genomic data post-processing. For FIO micro-benchmark, QZFS with QAT-accelerated gzip can achieve up to 5x average write throughput with a similar compression ratio (3.6) and about 80% reduction of CPU resource consumption (from more than 95% CPU utilization to less than 20% CPU utilization), compared to the software-implemented alternative. For practical genomic data post-processing workloads, benefiting from QAT acceleration, QZFS provides 65.83% reduction of average execution time and 75.58% reduction of CPU resource consumption over the software gzip implementation. Moreover, as compression acceleration is performed at the file system layer, QZFS also significantly outperforms the traditional simple gzip acceleration for applications while conducting genomic data post-processing.

2 Background and Motivation

This section presents data compression benefits and the motivation of hardware-assisted compression.

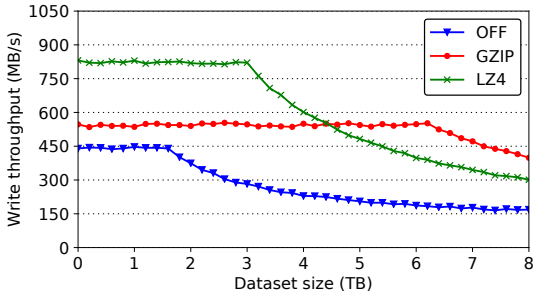


Figure 1: Write throughput on hybrid storage of one 1.6TB NVMe SSD and backup HDDs. Gzip and LZ4 achieve a compression ratio of about 3.8 and 1.9 respectively.

2.1 Data Compression on Storage Devices

As high-performance storage devices, NVMe SSDs can remarkably improve the read/write speed with low energy consumption [25,49]. Nonetheless, the limited capacity and high price significantly discourage their widespread use and storage devices have accounted for a large proportion of Total Cost of Ownership (TCO) [50]. In the Mistral Climate Simulation System, storage devices occupy more than 20% of the TCO for the entire system [28]. Many studies have investigated data compression on storage devices to improve I/O performance and reduce system TCO simultaneously [31,36].

To show the benefits of data compression, we evaluated the performance of a compression-enabled file system (i.e., ZFS) by the FIO tool [5] on a hybrid storage system, including one 1.6TB NVMe SSD (Intel® P3700 series) and backup HDDs. Two representative lossless compression algorithms, gzip [16] and LZ4 [35], were used in ZFS to compare with the compression OFF configuration. As shown in Figure 1, the write throughput with data compression (for both gzip and LZ4) outperforms the case of OFF because compression can effectively reduce the total data size written into the storage [33,53]. If the dataset size is larger than the capacity of the 1.6TB NVMe SSD, the excessive data are written into backup HDDs. Due to the poor read/write performance of HDD, the OFF configuration incurs throughput degradation rapidly once the dataset size exceeds 1.6TB. The gzip algorithm achieves a compression ratio of about 3.8 in this evaluation and the write throughput degrades after the dataset size exceeds 6.1TB. Since LZ4 is a fast compression algorithm (i.e. CPU time for compression is largely reduced), it can bring a higher write throughput than the gzip case although the compression ratio is lower, with a value of about 1.9. However, the performance of the LZ4 configuration begin to degrade after the dataset size exceeds 3TB and has no advantage over the gzip algorithm for a dataset size large than 4.5TB. In conclusion, data compression improves space efficiency and allows more data to benefit from the high-performance storage devices.

OFF表示不使用压缩

这段话就是在分析上面图2.1的，SSD为1.6T,之后就是往磁盘HDD中写入，因此IO性能开始下降了。

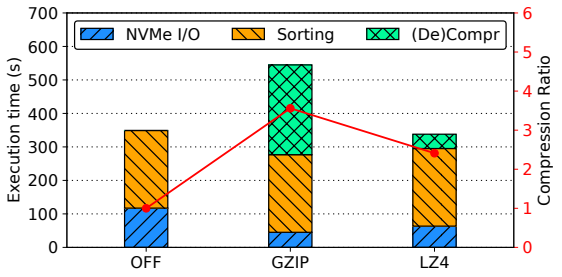


Figure 2: The execution time of genomic data sorting under different compression algorithms

2.2 The Selection of Compression Algorithms

Compression-enabled file systems leverage lossless data compression algorithms to transparently serve upper applications. An algorithm with both high compression ratio and low CPU resource consumption is an optimal choice yet these two aspects are actually hard to achieve at the same time. We conducted an experiment with scientific big data workloads running on ZFS to compare two representative lossless algorithm: gzip and LZ4. SAMtools [30], a popular set of utilities for sequence analysis, was used to manipulate 150GB genomic data stored on a NVMe SSD and perform the sorting operation, which needs to create temporary files and conduct a series of complex data reading and writing actions.

Figure 2 illustrates the breakdown of the total execution time under different compression algorithms, including *NVMe SSD I/O time (T_{io})*, *sorting time (T_s)* and *(de)compression time (T_c)*. The execution time for the compression OFF configuration only comprises of T_{io} (117.21s) and T_s (231.77s). The gzip and LZ4 algorithms have a similar T_s value because of the same sorting processing while T_{io} is reduced to 35.02s and 48.45s respectively due to the different compression ratios: 3.56 for gzip and 2.41 for LZ4. However, the high compression ratio of gzip leads to a high T_c value of 278.02s, compared to the 24.77s for the LZ4 configuration. This high CPU resource consumption of gzip may further cause resource competition and impact other tasks. Intuitively, if (de)compression operations can be offloaded to hardware accelerators to eliminate T_c for CPU, gzip could be an ideal compression algorithm as it can achieve the highest space efficiency. This motivates the design of a hardware-assisted compression-enabled file system for high-performance and cost-efficient data storage.

2.3 Hardware-Assisted Data Compression

Nowadays, diverse hardware accelerators are continuously emerging in cloud infrastructures and datacenters [10, 31, 32]. ASIC accelerators are increasingly attracting attentions due to their advantages on performance and energy-efficiency over general-purpose CPU, GPU and FPGA [1, 8, 32]. This paper selects Intel® QAT, a purpose-built ASIC for cryptography and compression, to offload (de)compression tasks and free up CPU resources. The latest high-performance QAT device has been directly integrated into chipsets and is becoming increasingly cheaper [22].

In essence, the offloading of a (de)compression task is to replace the compression-related function call with an I/O call to interact with the underlying QAT accelerator. However, the way the QAT hardware treats data (e.g., physical address and DMA operation) is different from that in the case of software-implemented compression (i.e., using CPU). Runtime translation and optimizations are necessary and important to achieve an efficient offloading. Moreover, considering the offload overhead (e.g., preparation/consumption of QAT requests/responses and PCIe transactions) and the needed preallocated system resources for QAT offloading, not all (de)compression tasks are worth being offloaded into QAT hardware. A well-designed heterogeneous data compression system should investigate the necessity of software/hardware switch or even compression/non-compression switch.

3 System Design

In this paper, based on ZFS file system and Intel® QAT compression accelerator, we propose QZFS (QAT-accelerated ZFS), which can serve as either a local file system or the back-end file system of Lustre (a type of parallel distributed file system). The overall QZFS architecture, including the I/O path of storage and the QAT acceleration subsystem, is illustrated in Figure 3. The modification starts with the ZIO Module of ZFS. The ZIO Module is a centralized I/O process module where all I/O requests are abstracted as ZIOs and forwarded to other modules for further processing, such as data compression and checksum verification. Among the subsequent modules, the ZIO_Compress Module is responsible for data (de)compression. To enable QAT offloading of (de)compression operations, QZFS introduces two new modules: the *Compression Service Engine* and the *QAT Offloading Module*.

To explain the functions of these QZFS modules, the compression workflow is depicted in Figure 3. **(1)-(2)**: The ZIO Module forwards ZIO requests to the ZIO_Compress Module which registers compression algorithms and delivers configuration information to the Compression Service Engine. **(3)**: The Compression Service Engine selects the compression algorithm among one (gzip) accelerated by QAT and others from software compression libraries. **(4)-(5)**: The QAT Offloading Module sends compression requests to the QAT accelerator and consumes QAT responses to fetch compressed result data. **(6)-(8)**: The compressed result data is returned and goes through the upper modules one by one. The data decompression workflow is similar to this compression workflow but does not involve the selection of decompression algorithm.

The main function of the Compression Service Engine is to serve as a selector of diverse compression algorithms. The

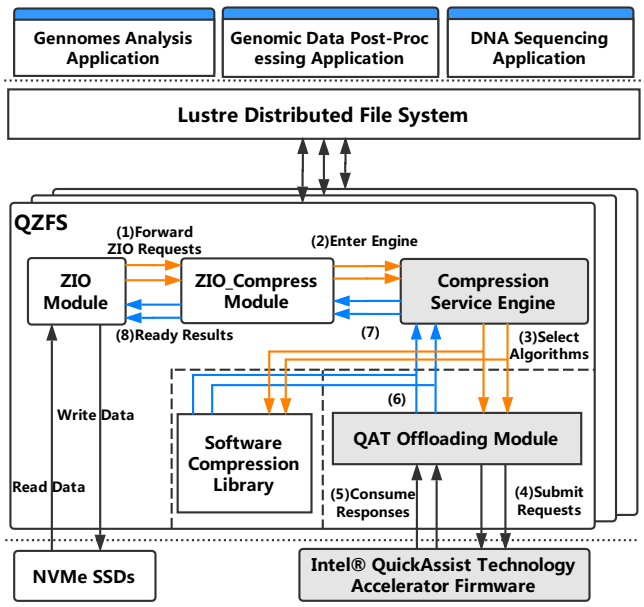


Figure 3: The overall architecture of QZFS

QAT-accelerated gzip is the default algorithm and another four software-implemented compression algorithms, including gzip, LZ4, ZLE and LZJB, are provided. The engine currently selects the compression algorithm according to the configuration set by users. Besides, the Compression Service Engine provides a uniform interface to the upper module. This kind of decoupling makes it able to be easily extended to support other hardware accelerators. Developers only need to focus on the detailed implementation of a new compression scheme and add it to this engine.

The QAT Offloading Module is responsible for offloading data (de)compression operations to the QAT accelerator. The existing source data prepared by the ZIO_Compress Module are suitable for software-based compression schemes. However, they cannot be directly used in the (de)compression requests to the QAT accelerator as data may be mapped into discontiguous physical memory and cannot be accessed through DMA operations of the QAT accelerator. To address this problem, a vectored (a.k.a. scatter/gather) I/O model is introduced to avoid memory copy and ensure that the QAT accelerator can sequentially read source data from multiple flat (i.e., physically contiguous) buffers and organize them to a single data stream for (de)compression, or read (de)compressed result data from a single data stream and organize them to multiple flat buffers.

为什么发给QAT的数据要特殊处理的原因

4 Implementation & Optimization

The implementation of QZFS prototype relies on QAT development APIs and Linux environment (CentOS with Linux Kernel 3.10) and it has been integrated into ZFS official releases [37]. This section introduces detailed features and recent bug fixes of QZFS, especially regarding performance optimizations, to demonstrate the accomplishment of effective compression offloading in QZFS.

4.1 Compression Service Engine

The detailed architecture of the Compression Service Engine is illustrated in Figure 4, which contains a *compressibility checker* and an *algorithm selector*.

4.1.1 Compressibility Dependent Offloading

Compressibility is a performance factor that is determined by data compression ratio to reduce unnecessary offloading operations. Low compressibility means that data are not worth being stored in a compressed format because, in that case, the compressed data will not save much storage space but only incur extra resource consumption for later decompression. Even with the QAT acceleration, the decrease of unnecessary offloading operations is beneficial as QAT resources can be spared for useful (i.e., high compression ratio) tasks, especially in peak hours.

The *compressibility checker* in QZFS uses an auto-rejection method to determine whether to store the data in a compressed format or not. When the checker receives a ZIO request, it has the knowledge of the source data size, denoted by S_{src}, and presets the result data size, denoted by S_{res}, for placing the returned compressed data. Conventionally, S_{res} may be set to the same size as S_{src} when executing compression tasks. In QZFS, the *compressibility checker* uses a default $S_{res} = 0.9 * S_{src}$ for QAT-accelerated gzip algorithm to indicate a compressibility threshold of 10%. If the compressed result data overflows the buffer of S_{res}, it is automatically rejected and the uncompressed source data is returned to the ZIO_Comress Module as the result. Compared to the original compressibility threshold (i.e., 12.5%) in ZFS, QZFS actually relaxes the restriction as the decompression operation can be performed more efficiently by the QAT accelerator. Users can further adjust this compressibility threshold to maximize space efficiency, depending on the characteristics of workloads and hardware conditions. For data decompression, the compressibility checking is not necessary and the S_{res} is set to the size of original uncompressed data, which is recorded by QZFS during compression processing.

4.1.2 Selective Offloading by Source Data Size

The source data size (i.e., S_{src}) is an important factor that may have an influence on system performance. ZFS has a parameter named *record size* which defines the maximum size of a block that may be compressed and written by ZFS. A storage I/O operation with data size smaller than the record size may be packed into one ZIO request for processing. That's to say,

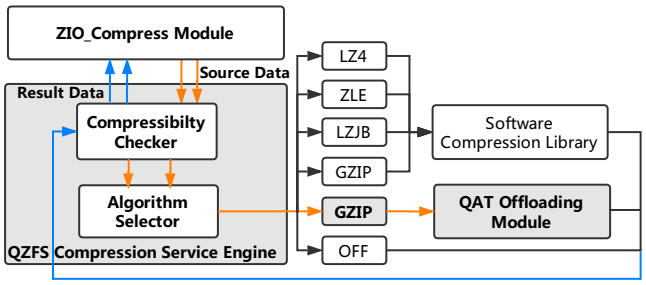


Figure 4: Architecture of compression service engine

S_{src} is variable within the upper limit defined by the record size. ZFS uses a default record size value of 128KB, which achieves a good performance in most cases, and users may modify this parameter to obtain better performance in their own scenarios.

没看懂

QZFS selectively offloads ZIO requests with S_{src} from 4KB to 1MB and uses software alternatives to process other ZIO requests. For small source data (i.e., $S_{src} < 4$KB), the offload overhead, including preparation/consumption of QAT requests/responses and PCIe communication, offsets most of the benefit of QAT accelerating, so the software-based compression is a better choice. The QAT support for big source data (i.e., $S_{src} > 1$MB) requires a large kernel memory preallocated to work as intermediate buffers. This memory size is several times the product of the maximum S_{src} and the number of allocated (typically tens of) QAT instances. Moreover, for a fixed number of worker threads in ZFS that synchronously offload (de)compression operations to QAT, the use of bigger source data cannot give obviously higher performance. Therefore, QZFS only preallocates a kernel memory region that can support a maximum S_{src} of 1MB to achieve both good performance and acceptable consumption of kennel memory resources.

4.1.3 Applicability and Availability

The *algorithm selector* is implemented as a centralized scheduler of data compression algorithms. Algorithms are organized in a scalable algorithm vector that can easily be extended to incorporate other algorithms, either software-based or hardware-assisted ones. Developers only need to focus on the detailed implementation of a new compression scheme, add it to this the algorithm vector and update the configuration as needed. When runtime error occurs in some hardware accelerator, the algorithm selector can seamlessly switch to other software alternatives to provide fault tolerance and high availability.

Currently, the selection of compression algorithm mainly relies the the configuration, which defines the priorities of different algorithms, with the QAT-accelerated gzip as the default one. If in future, a number of hardware-assisted compression algorithms with their own features are incorporated into the Compression Service Engine, an intelligent selection (e.g., use the hints from upper layers) is a good optimization to reap the strengths of different compression schemes.

4.2 QAT Offloading Module

4.2.1 Vectored I/O Model

The QAT accelerator accesses data blocks through DMA operations, which require the data to be stored in contiguous physical memory. The original source and result data of a ZIO request are stored using virtual memory pointed by two pointers P_{src} and P_{res}. The vectored I/O model can effectively bunch together the discontinuous memory to form I/O transactions for DMA operations. As illustrated in the left part of Figure 5, we employ two buffer structures, *flat buffer* and *scatter/gather buffer list* (SGL), to implement the vectored I/O model.

The flat buffer consists of two parts: a data buffer length, denoted by *DataLenInByte* and a pointer, *pData*, to the data buffer owning contiguous physical memory. SGL is introduced to organize multiple flat buffers in a vector manner and consists of four parts: (1)*numBuffers*, the number of flat buffers in this list; (2)*pBuffers*, a pointer to an unbounded array containing multiple flat buffers; (3) *pUserData*, an opaque field; (4)*pPrivateMetaData*, private representation of this buffer list. In summary, SGL describes a collection of flat buffers, each of which is physically contiguous. The QAT accelerator can parse the SGL structure to obtain the beginning physical address of each flat buffer and sequentially access each data block through DMA operations.

4.2.2 Data Reconstruction and Memory Zero Copy

A simple approach for data reconstruction is to allocate enough contiguous physical memory and copy data from/to this memory. Specifically, before the (de)compression offloading, a region of contiguous physical memory is allocated to store data copied from P_{src} and delivered in the request to QAT. Also, another region of contiguous physical memory is allocated to store the response (i.e., result data) from QAT. After the completion of (de)compression offloading, the result data is copied from the contiguous physical memory to P_{res} prepared by ZIO.

Although the allocated contiguous physical memory can be reused by multiple ZIO requests, this approach inevitably introduces the overhead of memory copy, which likely become a bottleneck in today's high speed I/O. Therefore, we introduce the vectored I/O model, along with virtual address translation and memory mapping, to achieve memory zero copy. As shown in Figure 5, the QAT Offloading Module translates the virtual addresses of the source data prepared by ZIO into physical addresses and organizes the physical contiguous data blocks into the Input SGL structure.

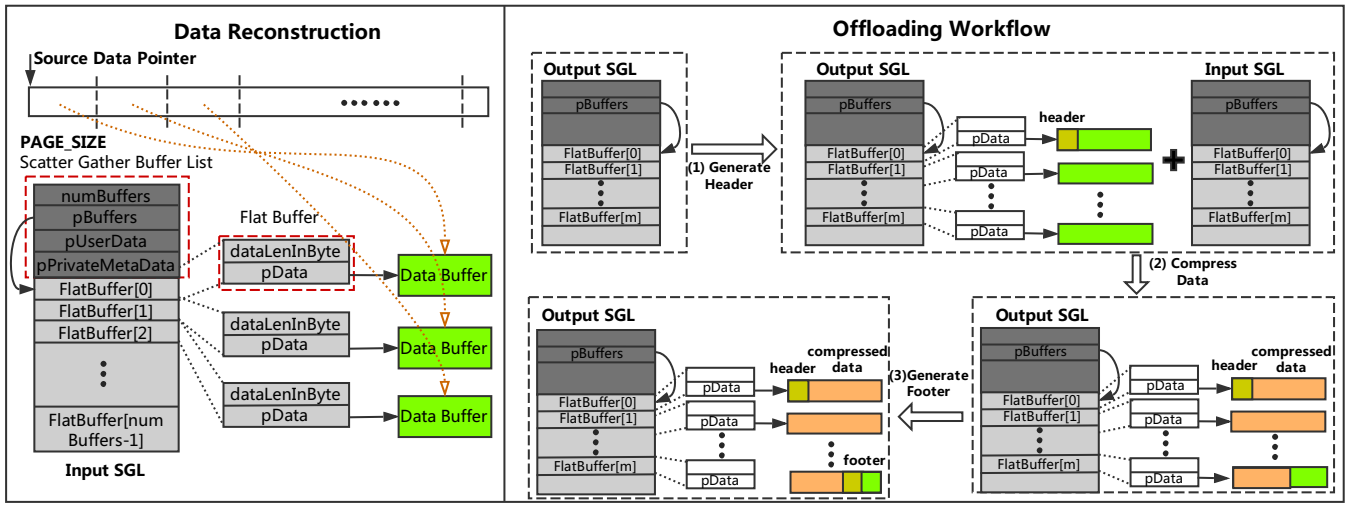


Figure 5: Data reconstruction and QAT offloading workflow in the QAT Offloading Module

A physical page (a.k.a. page frame) is the smallest fixed-length contiguous block of physical memory into which virtual pages are mapped by the operating system. The source data represented by contiguous virtual addresses could be divided into physical contiguous data blocks in terms of physical pages. The QAT Offloading Module directly maps these physical pages to serve as the data of the flat buffers and each flat buffer maps at most one *PAGE_SIZE* data. Note that the start and end virtual addresses of the source data may not be page-aligned, which has an influence on the number of involved physical pages. For example, in the case of 4KB PAGE_SIZE, a 11KB source data may correspond to four physical pages (2KB+4KB+4KB+1KB), instead of three ones. This information is important for the creation of the data structures of Input/Output SGLs. The memory allocation for the Input SGL data structure is performed according to the maximum possible number of flat buffers: $numBuffers = (S_{src} >> \text{PAGE\_SHIFT}) + 2$.

The details of the data reconstruction for source data (i.e., building Input SGL) are as follows and the process for the result data buffer to build Output SGL is similar. At first, the virtual address indicated by the P_{src} pointer (pointing to the start address of the source data initially) is translated to obtain the corresponding physical page structure. Note that the virtual address may come from different kernel memory zones, including the *vmalloc region* or the *direct memory region*. The QAT Offloading Module checks whether the a virtual address belongs to the vmalloc region by invoking the *is_vmalloc_addr* function. If so, the *vmalloc_to_page* function is used to get the corresponding physical page structure; otherwise, the *virt_to_page* function is used to obtain the right page. Next, the QAT Offloading Module employs the *kmap* function to establish a long-lasting mapping from kernel's address space to the obtained physical page. The *pData* field of the first flat buffer points to the returned mapped address, plus the same page offset as the P_{src} value. The *dataLenInBytes* field is accordingly set by considering the page offset. Finally, the P_{src} pointer moves to the beginning of the remaining untreated source data and the above steps are repeated to fill subsequent flat buffers. For the last piece of the source data that may correspond to only part of a physical page, the *dataLenInBytes* field is set to the actual size of this last piece. After the completion of an offloading task, the *kunmap* function needs to be invoked to release long-lasting mappings.

4.2.3 QAT Offloading Organization

Overflow avoidance and load balancing: When QZFS boots up, the QAT Offloading Module initializes the QAT logical instances to set up communication channels for requests/responses to/from the QAT accelerator. Thus, a region of contiguous physical memory needs to be allocated for the QAT instance which includes an *intermediate buffer* to place run-time process data (e.g., dynamic Huffman encoding) of the QAT accelerator. The size of the intermediate buffer should be enough for the maximally allowed source data (i.e., 1MB). Data compression is supposed to reduce data size but compression algorithms may cause data expansion at some moment during compression. To avoid the buffer overflow, its size is enlarged to be double of the maximally allowed data size. Besides, when too many data compression tasks are offloaded, the module may not obtain the QAT logical instances because there are not enough QAT computing resources. Therefore, the module will balance the system's computational resources by sending the task to software alternatives and employing the CPU to finish it.

Fail recovery: A QAT compression session describes the compression parameters to be applied across a number of requests. After the initialization of logical instances and source/result data reconstruction, the module sets up the com-

munication with the QAT accelerator by QAT compression sessions. Occasionally, the error may occur in these compression sessions, such as a driver process crash where a wrong DMA address passed to the accelerator. If a failure on the QAT accelerator cannot be handled correctly, the QAT device may be restarted for recovery. In this situation, the QAT Offloading Module cleans all sessions and sets an availability flag as FALSE to disable all QAT offloading actions until the completion of re-initialization. The related internal data structures, QAT logical instances and intermediate buffers will be reset as well.

Offloading workflow: The data compression offloading operations are organized as three steps shown in the right part of Fig. 5. **First**, a header generator API function is called to produce the gzip style header[1], which requires the output SGL as the parameter. The header is added to the front of the output SGL, and the *pData* of the flat buffer is moved to the corresponding offset of the gzip header. **Second**, the data compression API function submits input and output SGLs to the QAT accelerator that consumes the source data from the input SGL and generates processed result data to the output SGL. Note that the QAT accelerator uses the interrupt mode for response processing which requires the module to use the *wait_for_completion_interruptible_timeout* function to wait the completion of the tasks. **Third**, a gzip compliant footer[2] is produced by the footer generator API function. More details of the header and footer formats are given in RFC 1952 [13], which are supported by the QAT accelerator. The difference with data decompression is that it invokes the data decompression API function for three steps.

5 Evaluation

In this section, we first describe the evaluation platform and testing methodology. Then we use FIO micro-benchmark and scientific big data processing workloads to evaluate our implemented QZFS prototype on cluster servers.

5.1 Evaluation Methodology

Experimental testbed: We established an experimental testbed with four physical servers, each of which was equipped with two 22-core Intel® Xeon® E5-2699 v4 processors, 128GB RAM, one NVMe SSD array and one Intel® DH8950 PCIe QAT card [21]. The NVMe SSD array was comprised of three 1.6TB Intel® P3700 Series NVNe SSDs. In each server, a separate SATA SSD was used for housing the operating system (CentOS 7.2) with Linux Kernel 3.10 and QZFS. All these servers were connected via Intel® XL710 40GbE NICs and a 100GbE switch. The detailed topology is illustrated in Figure 6.

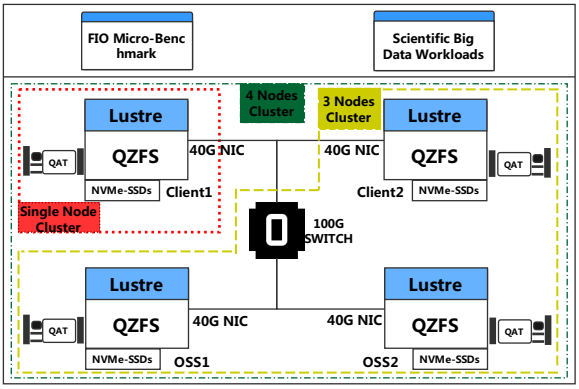


Figure 6: Topology of experimental testbed

Cluster: QZFS was deployed as the back-end file system of Lustre distributed file system. To evaluate different Lustre scenarios, three types of cluster settings were used: (1) all-in-one *single-node cluster*, i.e., a single physical server as both the client and the Object Storage Server (OSS); (2) *three-node cluster* with one physical server as the client and the other two physical servers as OSSes; (3) *four-node cluster* with one more physical server as a client (on the basis of the three-node cluster). All the servers in a cluster shared the same Lustre distributed file system. The client server ran benchmark workloads locally and read/wrote data from/to one or two OSSes.

Note that Lustre inherently provides the ability to support a large number of clients. In our in-lab evaluation, the number of available clients was limited. As a workaround, we leveraged high-performance physical servers along with high-speed NICs to work as *heavy* clients and produce enough stress. In the following experiments, we will show that the total stress is equivalent to tens to hundreds of ordinary clients. The I/O stress from an ordinary client depends on a lot of factors, including workload characteristics, client hardware limits (e.g., NIC limit), the interconnection between Luster clients and OSSes, etc. We assume that each ordinary client generates an I/O stress between 100Mbps and 1Gbps.

Performance metrics: We mainly compared three configurations: (1) *OFF*: QZFS without data compression; (2) *GZIP*: QZFS with software-implemented gzip algorithm enabled; (3) *QAT*: QZFS with QAT-accelerated gzip algorithm enabled. Other algorithms in the software compression library including LZ4, ZLE and LZJB may also be measured for comparison. The following important metrics were collected as performance indicators: read/write *throughput* of micro-benchmarks, average *execution time* of completing big data analytic tasks, *compression ratio* for indicating space efficiency and *CPU utilization* (collected using Intel® Performance Analysis Tool [23]) for indicating computing resource

[1] a gzip header indicates metadata including compression ID, file flags, timestamp, compression flags and operating system ID.

[2] a gzip footer containing a CRC-32 checksum and the length information of the original uncompressed data.

consumption.

Particularly, we define a new comprehensive metric to measure the *cost-efficiency* of the entire system, which equals compression ratio divided by CPU utilization (i.e., computing resource consumption):

$$\frac{compression\_ratio}{cpu\_utilization}$$

CPU and high-performance storage devices account for a large proportion of TCO. This metric reflects a combined benefit of saved storage space and saved CPU resources, and a higher value means more cost-efficiency gains. Note that we do not consider QAT cost in this metric because QAT device is becoming increasingly cheaper. The latest high-performance QAT, directly integrated into chipsets, only costs about $32 (comparing $132 for C625 chipset with QAT and $100 for C624 chipset without QAT) [22], which is negligible in comparison to CPU price.

5.2 Evaluation Benchmark

Micro-benchmark workloads: FIO (Flexible I/O tester) [5] is a productive tool in Linux that can simulate all kinds of I/O workloads and accurately measure I/O performance. We used FIO to spawn several threads or processes for measuring read/write throughput. The default FIO setting for random data generation and management were employed. More specifically, a buffer of random data (actually pseudo-random) was created at the beginning and it was used continuously during the test to reduce overhead. To comprehensively evaluate QZFS, we conducted the FIO experiments with different I/O patterns, block sizes (i.e, the size of chunks for issuing read/write I/O) and compression algorithms.

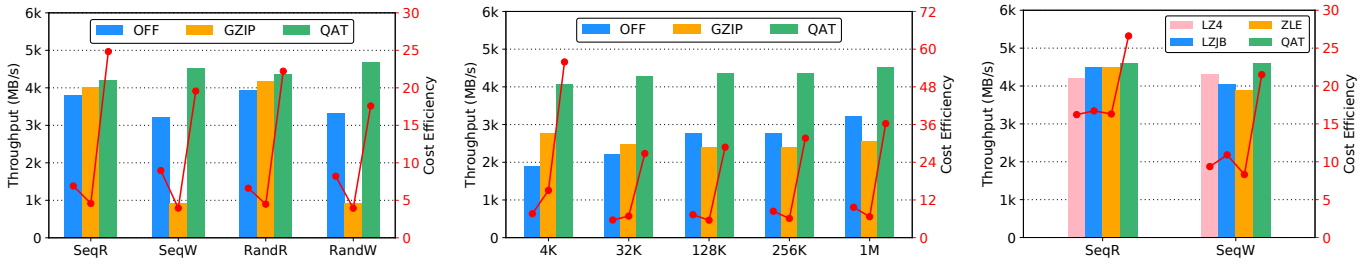
Scientific big data workloads: Genomic data post-processing is a representative workload of scientific big data. The original 3TB genomic dataset used in this experiment are available in European Nucleotide Archive [29] and International Genome Sample Resource [11]. These data are stored as specific formats, such as FastQ, BAM and SAM, which are widely used in both industry and academia. For each client, two genomic data post-processing tools, SAMTools (v1.3.1) with htslib (1.3.2) [30] and Biobambam2 (v2.0.82) with libmaus (2 2.0.435) [47], were used to work as computing workloads. Specifically, we selected five representative operations from these two tools to evaluate the advantage of QZFS. *Converting* is to convert one kind of genomic data to other kind (e.g. FastQ format converted to BAM format in this evaluation), which involves heavy CPU-bound tasks. *Viewing* is to print all alignments information in the specified input file to standard output in SAM format (with no header). *Sorting* is to sort these data by leftmost coordinates and create temporary BAM files as needed when the entire alignment data cannot fit into memory. *Merging* is to merge multiple sorted files, producing a single sorted output file that contains all the input records and maintains the existing sorting order. *Indexing* is to index a coordinate-sorted BAM or CRAM file for fast random access.

5.3 FIO Micro-benchmark

The experiments were conducted in the four-node cluster where 16 FIO threads were created in each Lustre client to read/write job files independently from/to the two Lustre OSSes. The total size of file I/O for each FIO thread is 2GB. The read/write throughput stated in the evaluation is the sum value collected from two clients and the CPU utilization is the average value collected from Lustre OSSes.

Figure 7a shows experimental results for diverse I/O patterns with a fixed 128KB FIO block size, including sequential read/write (SeqR/SeqW) and random read/write (RandR/RandW). In the compression OFF configuration, the average read throughput (i.e., the average value of SeqR and RandR) is 18% higher than the average write throughput, and RandR achieves up to 3937 MB/s throughput, which is the highest. These results are in accordance with the hardware characteristics of NVMe SSDs [17]. The average read throughput in the GZIP configuration outperforms the average write throughput by about 4.5x because the gzip algorithm has an asymmetric compression/decompression speed, with the former lagging much behind the latter [1]. After enabling the QAT accelerator, the average read throughput is similar with the average write throughput. In general, the QAT configuration has the highest read/write throughput and cost-efficiency. The GZIP configuration has a similar compression ratio with the QAT configuration (3.78:3.65), but its high CPU resource consumption (i.e., long compression time) causes not only low write throughput but also low cost-efficiency. The QAT offloading for gzip compression operations achieves about 5x improvement on average write throughput and enhances the cost-efficiency by a factor of more than four. In comparison to the OFF configuration, the QAT configuration also provides a throughput improvement (10% for read and 28% for write) as the compressed data reduces storage I/O cost. Meanwhile, the cost-efficiency is enhanced by a factor of more than three due to the high compression ratio (3.65).

Figure 7b shows the I/O throughput and cost-efficiency with the same amount of sequential read and write operations (SeqR:SeqW = 1:1) while varying the FIO block size in each I/O request from 4KB to 1MB. Note that the record size (128KB by default) in ZFS only defines the maximum size of a block that may be compressed and written by ZFS. As a result, a FIO block may be directly compressed/written by ZFS or multiple FIO blocks may be combined into a whole block for processing. For the compression OFF configuration which can directly benefit from the high-performance NVMe SSDs, the I/O throughput gradually grows from 1899MB/s to 3213MB/s (about 70% improvement) as the increase of FIO block size. For the GZIP and QAT configurations, the



(a) Different I/O patterns (b) Different block sizes (c) Different compression algorithms

Figure 7: The read/write throughput and cost-efficiency on FIO micro-benchmark

varying of FIO block size does not show obvious influence on I/O throughput. Still, the QAT configuration always has the highest I/O throughput and cost-efficiency. A notable phenomenon is that for the case of 4KB FIO block size with software or QAT-accelerated gzip in ZFS, the cost efficiency (actually compression ratio) is obviously higher than other cases with bigger FIO block sizes. This may be a joint result of the record size mechanism in ZFS and the reuse of random data in FIO. Specifically, multiple small (i.e., 4KB) FIO blocks may have a higher probability of being combined into a whole block in ZFS for compression. Then, the possible reuse of random data across these FIO blocks leads to a high compression ratio for the whole block.

Figure 7c compares the QAT-accelerated gzip with other software-implemented fast compression algorithms including LZE, LZJB and ZLE. The QAT configuration gains an average of 12.71% throughput improvement with SeqW and an average of 4.83% throughput improvement with SeqR, compared to these software-implemented algorithms. Also, QAT-accelerated gzip in ZFS provides an average of 2.25x and 1.62x cost-efficiency for SeqW and SeqR respectively. This performance advantage comes from the high compression ratio of the gzip algorithm and the offloading of gzip (de)compression operations. In addition, we can see that SeqR operations show an obviously higher cost-efficiency than SeqW operations. This is because decompression typically costs less computing resources in comparison to compression.

Finally, we give a calculation about how many ordinary clients the total stress in FIO experiments is equivalent to. As shown in Figure 7, the average stress from the two heavy client servers is more than 4000MB/s in the QAT configuration, which equals the stress from 32 to 320 ordinary clients (100Mbps to 1Gbps each).

5.4 Scientific Big Data Evaluation

The scientific workloads running on Luster clients interact with the Luster OSSes to access the stored scientific data. We used two different deployment modes for evaluation: shared deployment mode (i.e., single-node cluster) and separate deployment mode (including three-node and four-node clusters).

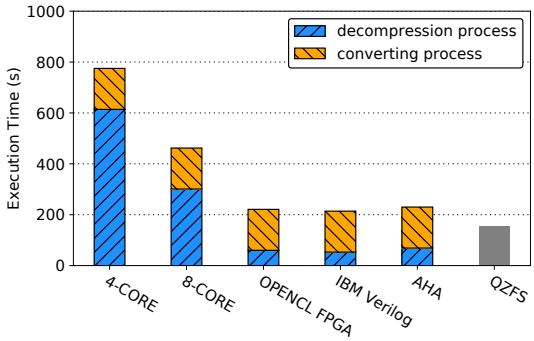
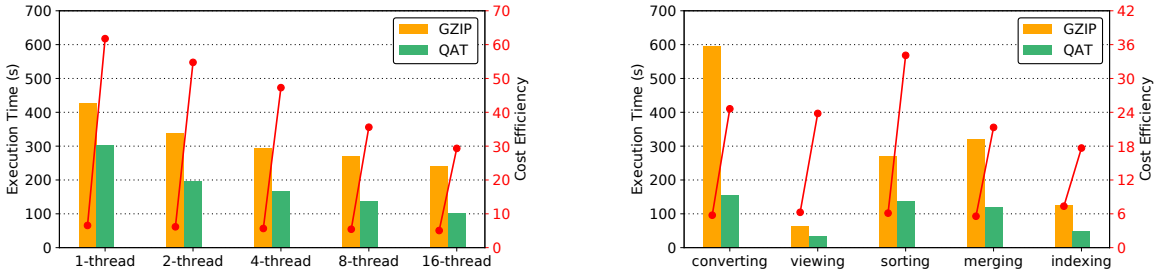


Figure 8: Execution time of a complete converting operation under different data compression schemes

5.4.1 Shared Deployment Mode

Eight scientific workload processes ran in parallel in the shared server. Each workload process was set to read/write 9.5GB data and a total of 76GB data may be processed simultaneously in memory.

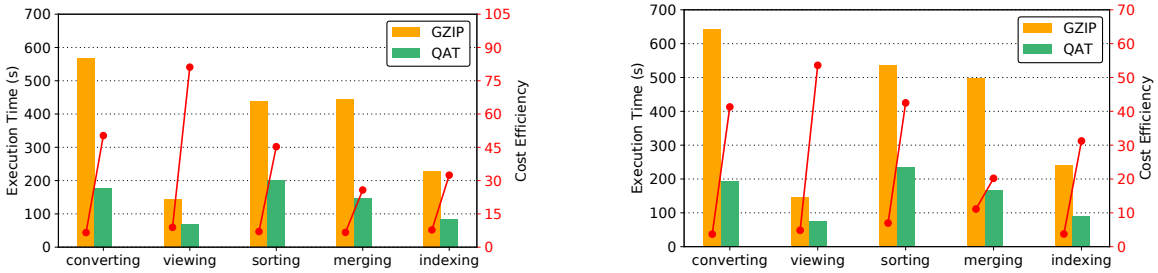
We first validated the benefits of QAT-accelerated gzip compression at the file system layer over *simple* gzip use at the application layer (i.e., the compressed big data needs to be first decompressed into storage and then read again for processing). The experiment results are shown in Figure 8. For the first five configurations, the execution time of a complete converting operation consists of two parts: (1) decompression process time with given computing resources (e.g., 4 cores or AHA accelerator), which includes the time for reading the compressed data from storage, the time for decompression and the time for writing the uncompressed data into storage; (2) converting process time using eight scientific workload processes, which includes the time for reading uncompressed data from storage (76GB totally), the time for converting operations and the time for writing new format data back to storage.



(a) Sorting performance with varying thread #

(b) Different operations under 8 threads

Figure 9: The execution time and cost-efficiency on shared mode deployment (single-node cluster)



(a) Three-node cluster with one client and two OSSes

(b) Four-node cluster with two clients and two OSSes

Figure 10: The execution time and cost-efficiency on separate mode deployment

The first five configurations share a same converting process time of 161.12s. The decompression process time of OpenCL FPGA, IBM Verilog and AHA is calculated according to the performance results reported in [1]. These accelerators show an obvious advantage on decompression compared to the 4 or 8 CPU cores. QZFS keeps data compression transparent to scientific workloads and obtains the shortest execution time (156.71s). This value is even smaller than the above pure converting process time without (de)compression (161.12s) because QZFS directly reads compressed data from storage, performs decompression and converting simultaneously and then writes compressed data back to storage. Since the reading/writing of compressed data largely reduces storage I/O cost and (de)compression is offloaded to QAT accelerator, a higher performance (i.e., shorter execution time) is achieved.

It is true that an application may integrate (de)compression module that can efficiently process compressed big data (e.g., small block based (de)compression and multiple threads), along with the enabling of QAT acceleration, to achieve similar performance to QZFS. However, this likely involves heavy modifications for each new application. In comparison, one-time modification to the file system (i.e., the proposed QZFS) can transparently benefit all applications running on it.

Figure 9a shows the execution time and cost-efficiency with varying number of threads in each scientific workload process performing sorting operations. When there is only one thread in each workload process, the software gzip configuration gives an execution time of 426s and a cost-efficiency value of 6.6. The QAT configuration provides a 30% reduction of execution time and enhances the cost-efficiency by a factor of nearly 10 due to the faster (de)compression and largely reduced CPU resource consumption. As the thread number increases from 1 to 16, the execution time decreases gradually for both GZIP and QAT configurations. In the case of 16 threads in each workload process, the performance advantage of QAT-accelerated gzip grows from 30% to 60% as more threads produce more parallel I/O requests and increases the utilization of underlying QAT accelerators. This further leads to a growing of CPU utilization for sorting from 5.8% to 12.2% in the QAT configuration. In comparison, the CPU utilization in the GZIP configuration grows from 64% to 83%.

Figure 9b evaluates the post-processing scientific workloads performing different operations with a fixed eight threads in each workload process. All the five types of operations need to read the genomic data from storage and the operations excluding viewing further needs to write the newly-generated data (smaller or bigger) back to storage. We can see that the QAT configuration achieves 73% and 63% reduction of execution time for the converting and merging operations respectively over the GZIP configuration. For other opera-

tions, the QAT-accelerated gzip in ZFS can provide about 2x performance enhancement. On average, the QAT configuration brings a 3.91x cost-efficiency improvement over the software-implemented gzip. Especially, the cost-efficiency enhancement is up to 5.57x for the sorting operation because it is a complicated operation that creates many intermediate data files and repeatedly involves reading (decompression) and writing (compression) actions.

The total stress from the one heavy client (eight threads in each workload process) is up to 75GB/35s=2143MB/s (the viewing operation), which equals the stress from 17 to 171 ordinary clients (100Mbps to 1Gbps each).

5.4.2 Separate Deployment Mode

We first evaluated QZFS with one client that remotely accesses two OSSes and then increased the client number to two. In each client sever, eight scientific workload processes ran in parallel and each workload process with a fixed eight threads was set to read/write 9.5GB data.

In the three-node cluster with different operations, as shown in Figure 10a, the QAT configuration on average reduces 63.10% execution time and achieves more than 6x cost-efficiency compared with the QAT configuration. Like the shared deployment mode, the QAT-accelerated gzip in ZFS still shows a high performance enhancement (68.95% reduction of the execution time) for the converting operation. A notable phenomenon is that the biggest cost-efficiency enhancement (9.11x) is witnessed on the viewing operation because it does not need to write data back to the remote storage (i.e., reduced CPU resource consumption on the costly network stack).

For the case of four-node cluster with double stress from two heavy clients, as shown in Figure 10b, the overall performance results are similar to the case of three-node cluster. In comparison to the software-implemented gzip in ZFS, the QAT acceleration on average provides a 63.14% reduction of execution time and a 6.26x improvement of cost-efficiency. It demonstrates that QZFS has a good scalability to provide stable and effective compression services as the scaling of clients. The total stress from the two heavy clients is up to 150GB/75s=2000MB/s (the viewing operation), which equals the stress from 16 to 160 ordinary clients (100Mbps to 1Gbps each).

6 Bottleneck Analysis

This section analyzes the performance bottleneck in QZFS. We use the experiment result of RandW in Figure 7a for analysis, which shows the highest I/O throughput (4680MB/s) for the QAT configuration. In this case, the average CPU utilization in the two OSSes is 20.2%, which means CPU resources are still abundant. The compression ratio is 3.55 and the actual NVMe SSD I/O throughput is 4680/3.55 = 1318MB/s, which means the NVMe SSD is not the bottleneck as the compression OFF configuration can achieve a storage I/O throughput of 3314MB/s. The network throughput for each physical server is about 4680/2 = 2340MB/s = 18.72Gpbs, which is only half of the hardware limit of 40GbE NIC. The QAT compression throughput in each Lustre OSS is also about 18.72Gpbs, which reaches nearly 80% of the hardware limit (24Gpbs [21]) of a DH8950 QAT card. In summary, the I/O throughput (4680MB/s) does not achieve the hardware limit of the testbed system while this throughput cannot be further enhanced even if we launch more FIO threads in clients to generate more RandW jobs in parallel.

Actually, the main performance bottleneck may reside in the ZFS software stack. Although ZFS is designed to automatically leverage multi-core resources, the number of ZFS worker threads that can offload (de)compression operations to QAT is restricted by: the number of CPU cores and the number of available QAT instances. What's more, the worker thread interacts with QAT in a synchronous mode, which means the worker cannot submit the next (de)compression request until the completion and consumption of the first one. As a result, the use of more FIO threads at the application layer cannot give rise to more concurrent/parallel (de)compression requests sent to QAT.

An approach to overcome this bottleneck is to optimize the way the worker thread interacts with QAT to increase the utilization of the underlying QAT accelerator. QAT provides an inherently non-blocking interface with the request/response mechanism. If we can enable asynchronous offload mode in ZFS, a single worker thread then has the ability to concurrently offload multiple (de)compression operations to QAT and the QAT hardware limit can be easily reached with only several threads. However, the enabling of asynchronous offload mode is a hard work, involving the design of (1) asynchronous support in all layers of ZFS software stack to correctly handle an uncompleted (de)compression block and (2) efficient pause (context saving) and resumption (context restoring) of an offload job. One can refer to our previous work [20] on how to enable high-performance asynchronous crypto offload framework for TLS servers/terminators.

7 Related Work

Hardware-assisted data compression techniques has been well studied in the literature by using general-purpose accelerators. Patel *et al.* proposed a parallel algorithm and implemented a bzip2-like lossless data compression scheme on GPU architecture [39]. Their implementation enabled the GPU to become a co-processor of data compression, which lightened the computing burden of CPU by using idle GPU cycles. Ozsoy *et al.* [38] presented a pipelined parallel LZSS compression algorithm for GUGPU. Li *et al.* proposed an efficient, scalable GPU-accelerated OLAP system which tackled the bandwidth discrepancy using compression and an optimized

data transfer path [31]. Abdelfattah *et al.* utilized the Open Computing Language to implement high-speed gzip compression on an FPGA, which achieved higher throughput over standard compression benchmark, with equivalent compression ratio [1]. Kim *et al.* presented high throughput Xpress FPGA compressor to achieve CPU resource saving and high power efficiency [26]. Fowers *et al.* detailed a scalable fully pipelined FPGA accelerator that performs LZ77 compression and static Huffman encoding at rates up to 5.6 GB/s [15]. Pekhimenko *et al.* employed FPGA and GPU accelerators to implement Base-Delta encoding data compression algorithms in stream processing [40]. In comparison, our QZFS leverages the emerging ASIC accelerator for compression offloading and integrates it into the layer of file system to provide transparent, high-performance and cost-efficient compression service.

8 Conclusions

High storage I/O performance and low Total Cost of Ownership (TCO) are two important optimization objectives, which are hard to be obtained at the same time. Data compression is considered as an effective solution, but the compression tasks incur high computing resource consumption and likely impact the running of application workloads. Instead of directly accelerating compression tasks in applications, this paper investigated the data compression offloading at the file system level. QZFS (QAT accelerated ZFS) was proposed to integrate Intel® QAT accelerator into ZFS file system to provide application-agnostic and cost-efficient data storage. The evaluation has validated that QZFS can effectively save CPU resources and further enhance the performance of big data processing workloads in comparison with the software-implemented gzip in ZFS or traditional gzip acceleration for applications.

Acknowledgments

The authors would like to thank the shepherd, Dr. Patrick P. C. Lee, and the anonymous reviewers for their valuable comments. This work is supported in part by the National Key Research and Development Program of China (No. 2016YFB1000502) and the National Science Fund for Distinguished Young Scholars (No. 61525204). Jian Li is the corresponding author.

References

[1] Mohamed S Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a chip: High performance lossless data compression on fpgas using opencl. In *Proceedings of the International Workshop on OpenCL (IWOCL)*, pages 4:1–4:9, 2014.

[2] AHA. Aha378: 80.0 gbps gzip compression/decompression accelerator, 2016. http://www.aha.com/DrawProducts.aspx?Action=GetProductDetails&ProductID=41.

[3] Apache. Apache hadoop, 2018. https://hadoop.apache.org/.

[4] Apache. Apache spark: Lightning-fast unified analytics engine, 2018. https://spark.apache.org/.

[5] Jens Axboe. Welcome to fio's documentation, 2017. https://fio.readthedocs.io/en/latest/.

[6] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In *Proceedings of the 2nd Usenix Conference on File and Storage Technologies (FAST)*, volume 215, 2003.

[7] Paolo Bruni, Maria Kroos Boisen, Gianmauro De Marchi, and Franco Pinto. Reduce storage occupancy and increase operations efficiency with ibm zenterprise data compression. Technical report, 2018.

[8] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, pages 7:1–7:13, 2016.

[9] Yanpei Chen, Archana Ganapathi, and Randy H Katz. To compress or not to compress-compute vs. io tradeoffs for mapreduce energy efficiency. In *Proceedings of the first ACM SIGCOMM workshop on Green networking*, pages 23–28, 2010.

[10] Eric S Chung, Peter A Milder, James C Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *Proceedings of the 43rd International Symposium on Microarchitecture (MICRO)*, pages 225–236, 2010.

[11] Laura Clarke, Susan Fairley, Xiangqun Zheng-Bradley, Ian Streeter, Emily Perry, Ernesto Lowy, Anne-Marie Tassé, and Paul Flicek. The international genome sample resource (igsr): A worldwide collection of genome variation incorporating the 1000 genomes project data. *Nucleic acids research*, 45(D1):D854–D859, 2016.

[12] Constantinos Costa, Georgios Chatzimilioudis, Demetrios Zeinalipour-Yazti, and Mohamed F Mokbel. Efficient exploration of telco big data with compression and decaying. In *Proceedings of the 33rd International Conference on Data Engineering (ICDE)*, pages 1332–1343, 2017.

[13] Peter Deutsch. Gzip file format specification version 4.3. Technical report, 1996.

[14] EMC. Emc data compression: A detailed review. Technical report, 2010. https://www.emc.com/collateral/hardware/white-papers/h8045-data-compression-wp.pdf.

[15] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. A scalable high-bandwidth architecture for lossless compression on fpgas. In *Proceedings of the 23rd International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 52–59, 2015.

[16] Jean-Loup Gailly and Mark Adler. The gzip home page, 2018. https://www.gnu.org/software/gzip/.

[17] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys (CSUR)*, 37(2):138–163, 2005.

[18] Christian Horn. A look at vdo, the new linux compression layer, 2018. https://www.redhat.com/en/blog/look-vdo-new-linux-compression-layer.

[19] HPE. Hpe 3par storeserv storage, 2019. https://www.hpe.com/us/en/storage/3par.html.

[20] Xiaokang Hu, Changzheng Wei, Jian Li, Brian Will, Ping Yu, Lu Gong, and Haibing Guan. Qtls: high-performance tls asynchronous offload framework with intel® quickassist technology. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 158–172, 2019.

[21] Intel. Product brief: Intel® quickassist adapter 8950. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/quickassist-adapter-8950-brief.pdf, 2015.

[22] Intel. Compare intel® products. https://ark.intel.com/content/www/us/en/ark/compare.html?productIds=97341,97342, 2019.

[23] Intel. Intel® platform analysis technology. https://software.intel.com/en-us/intel-platform-analysis-technology, 2019.

[24] Intel. Intel® quickassist technology (intel® qat), 2019. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html.

[25] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. Nvmedirect: A user-space i/o framework for application-specific optimization on nvme ssds. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2016.

[26] Joo Young Kim, Scott Hauck, and Doug Burger. A scalable multi-engine xpress9 compressor with asynchronous data transfer. In *Proceedings of the 22rd International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 161–164, 2014.

[27] Yannis Klonatos, Thanos Makatos, Manolis Marazakis, Michail D Flouris, and Angelos Bilas. Transparent online storage compression at the block-level. *ACM Transactions on Storage (TOS)*, 8(2):5:1–5:33, 2012.

[28] Michael Kuhn, Julian Kunkel, and Thomas Ludwig. Data compression for climate data. *Supercomputing Frontiers and Innovations*, 3(1):75–94, 2016.

[29] Rasko Leinonen, Ruth Akhtar, Ewan Birney, Lawrence Bower, Ana Cerdeno-Tárraga, Ying Cheng, Iain Cleland, Nadeem Faruque, Neil Goodgame, Richard Gibson, et al. The european nucleotide archive. *Nucleic acids research*, 39(suppl_1):D28–D31, 2010.

[30] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, and Richard Durbin. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[31] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. Hippogriffdb: balancing i/o and gpu bandwidth in big data analytics. *Proceedings of the VLDB Endowment*, 9(14):1647–1658, 2016.

[32] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. Asic clouds: specializing the datacenter. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, pages 178–190, 2016.

[33] Thanos Makatos, Yannis Klonatos, Manolis Marazakis, Michail D Flouris, and Angelos Bilas. Using transparent compression to improve ssd-based i/o caches. In *Proceedings of the 5th European conference on Computer systems (EuroSys)*, pages 1–14, 2010.

[34] Marvell. Nitrox iii security processor family, 2019. https://www.marvell.com/security-solutions/nitrox-security-processors/nitrox-iii/index.jsp.

[35] Takayuki Matsuoka. Lz4 - extremely fast compression, 2019. https://lz4.github.io/lz4/.

[36] Sangwhan Moon, Jaehwan Lee, Xiling Sun, and Yangsuk Kee. Optimizing the hadoop mapreduce framework with high-performance storage devices. *The Journal of Supercomputing*, 71(9):3525–3548, 2015.

[37] OpenZFS. Zfs hardware acceleration with qat, 2018. http://open-zfs.org/wiki/ZFS_Hardware_Acceleration_with_QAT.

[38] Adnan Ozsoy, Martin Swany, and Arun Chauhan. Pipelined parallel lzss for streaming data compression on gpgpus. In *Proceedings of the 18th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 37–44, 2012.

[39] Ritesh A Patel, Yao Zhang, Jason Mak, Andrew Davidson, and John D Owens. Parallel lossless data compression on the gpu. In *Proceedings of the Innovative Parallel Computing (InPar)*, 2012.

[40] Gennady Pekhimenko, Chuanxiong Guo, Myeongjae Jeon, Peng Huang, and Lidong Zhou. Tersecades: efficient data compression in stream processing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 307–320, 2018.

[41] Weikang Qiao, Jieqiong Du, Zhenman Fang, Libo Wang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. High-throughput lossless compression on tightly coupled cpu-fpga platforms. In *Proceedings of the 26th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 37–44, 2018.

[42] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9:1–9:32, 2013.

[43] Khalid Sayood. *Introduction to data compression*. Morgan Kaufmann, 2017.

[44] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, pages 380–386, 2003.

[45] Evangelia Sitaridi, Rene Mueller, Tim Kaldewey, Guy Lohman, and Kenneth A Ross. Massively-parallel lossless data decompression. In *Proceedings of the 45th International Conference on Parallel Processing (ICPP)*, pages 242–247, 2016.

[46] Jon Tate, Christian Burns, Bosmat Tuv-El, and Jorge Quintal. Ibm real-time compression in ibm san volume controller and ibm storwize v7000. Technical report, 2018.

[47] German Tischler and Steven Leonard. biobambam: tools for read pair collation based algorithms on bam files. *Source Code for Biology and Medicine*, 9(1):13, 2014.

[48] Wikipedia. Zfs, 2019. https://en.wikipedia.org/wiki/ZFS.

[49] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Manu Awasthi, Tameesh Suri, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance characterization of hyperscale applicationson on nvme ssds. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 473–474, 2015.

[50] Zhengyu Yang, Manu Awasthi, Mrinmoy Ghosh, and Ningfang Mi. A fresh perspective on total cost of ownership models for flash storage in datacenters. In *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 245–252, 2016.

[51] Xuebin Zhang, Jiangpeng Li, Hao Wang, Kai Zhao, and Tong Zhang. Reducing solid-state storage device write stress through opportunistic in-place delta compression. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 111–124, 2016.

[52] Bin Zhou, Hai Jin, and Ran Zheng. A high speed lossless compression algorithm based on cpu and gpu hybrid platform. In *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 693–698, 2014.

[53] Hongbo Zou, Yongen Yu, Wei Tang, and Hsuan-wei Michelle Chen. Improving i/o performance with adaptive data compression for big data applications. In *Proceedings of IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, pages 1228–1237, 2014.