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

Xiaokang Hu[1,2], Fuzong Wang[1,2], Weigang Li[2], Jian Li[1], Haibing Guan[1]

[1] *Shanghai Jiao Tong University* [2] *Intel Asia-Pacific R&D Ltd.*

Background

High-performance Computing (HPC)

- Powerful computing capabilities for handling Big Data
- Massive storage I/O read/write operations
- Requirement: performance and efficiency of the storage subsystem

Background

NVMe SSDs

NVMe Storage Array

High-performance Computing (HPC)

- Remarkable increase of read/write speed with low energy consumption
- **But:** high price
  - Intel® SSD Data Center Family: nearly $500/TB
  - *Mistral*, HPC system for climate research in Germany: storage subsystem accounts for roughly 20% of TCO.

Background

NVMe SSDs

High-performance Computing (HPC)

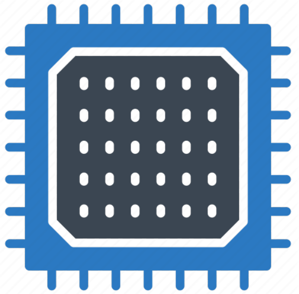
- 1st benefit: space efficiency → lower TCO
- 2st benefit: reduced I/O ops → higher performance
- **But:** at the expense of CPU resources

Background

NVMe SSDs

High-performance Computing (HPC)

Offloading to free up CPU



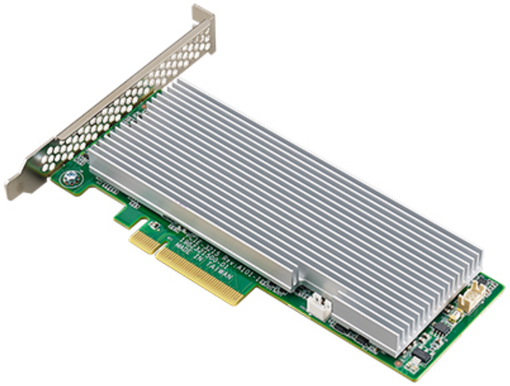
Accelerators (GPU, FPGA, ASIC)

Data Compression Acceleration

- **Compression in different system layers**

Application layer	File system layer	Block layer
• most common • e.g., Nginx, Hadoop	• benefit all applications • e.g., ZFS, BTRFS	• file system agnostic • e.g. RedHat VDO

Our work: ASIC-based compression offloading

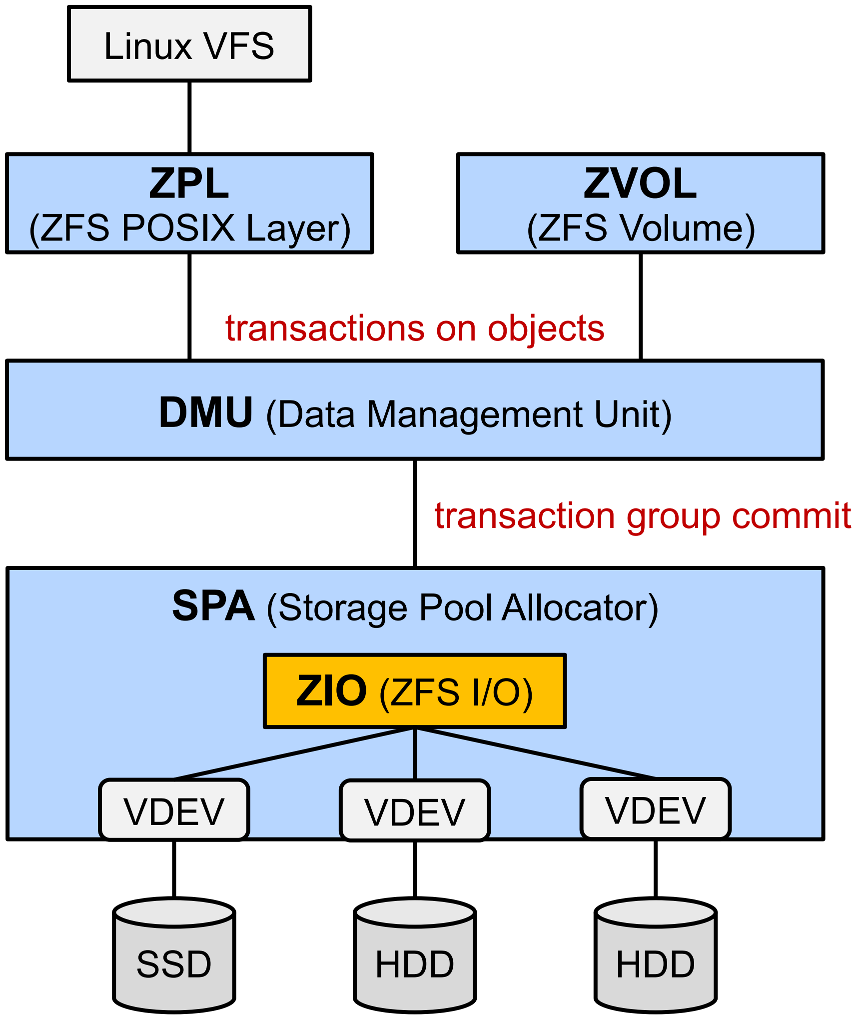


Intel® QuickAssist Technology (QAT)

- Modern ASIC for cryptography and compression
- Type: PCIe adapter, chipset, SOC
- Performance: up to 100Gbps
- Price: low to $32 after put into chipset

ZFS File System

- **ZFS Features**
  - Roles of both file system and volume manager
  - Pooled storage (no antique notion of volumes)
  - Transactional operation (always consistent)
  - End-to-end data integrity
  - RAID, encryption, compression, …
- **ZFS Record Size**
  - Define the max size (128KB by default) of a block that can be processed by ZFS
  - Varied block size for compression: 4KB, 36KB, 70KB, 128KB, …



QZFS (QAT-Accelerated ZFS)

- **Features**
  - Integration of Intel® QAT into ZFS for efficient data compression (***gzip*** algo) offloading

Transparently benefit ALL apps

High performance

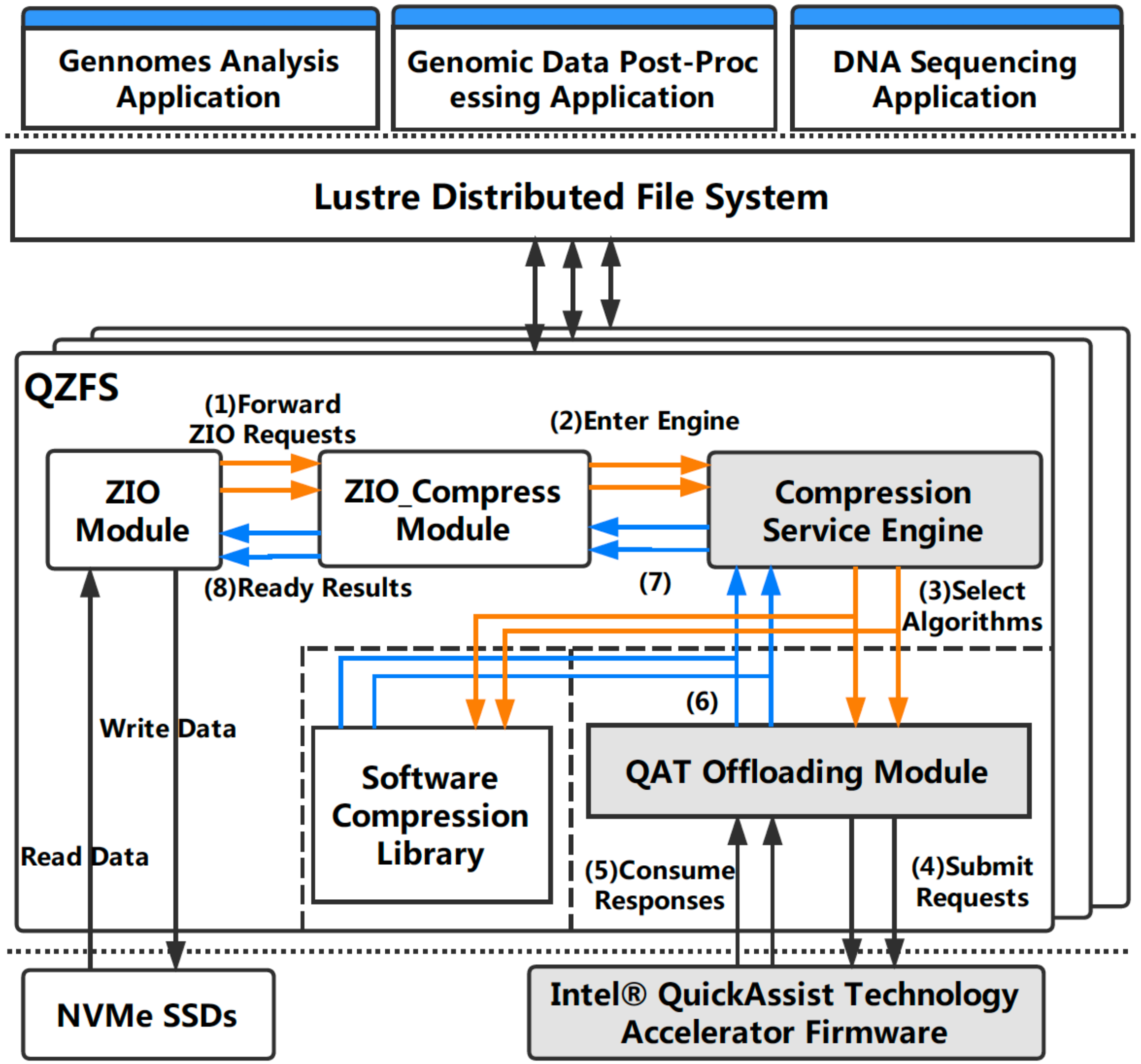
High space efficiency

Low CPU utilization

Cost efficiency

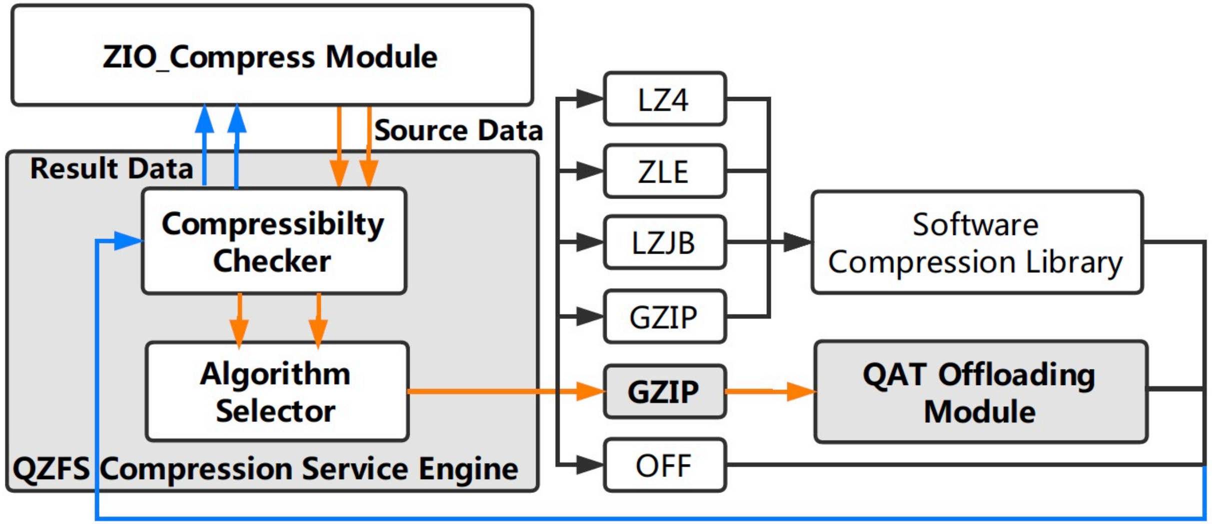
- **Design considerations**
  - Compression-related function → I/O call to interact with QAT
    - QAT HW treats data (i.e., physical address and DMA), different from SW (i.e., virtual address)
  - Offload overhead; pre-allocated system resources for QAT offloading
    - HW/SW switch
    - Compression/non-compression switch

QZFS Architecture



- **QZFS role**
  - local file system
  - back-end of Lustre (distributed)
- **ZIO module**
  - I/O requests are abstracted as ZIOs
  - ZIOs are forwarded to other modules
- **ZIO_Compress module**
  - data compression and decompression
- **Two new modules**
  - Compression Service Engine
  - QAT Offloading Module

Compression Service Engine



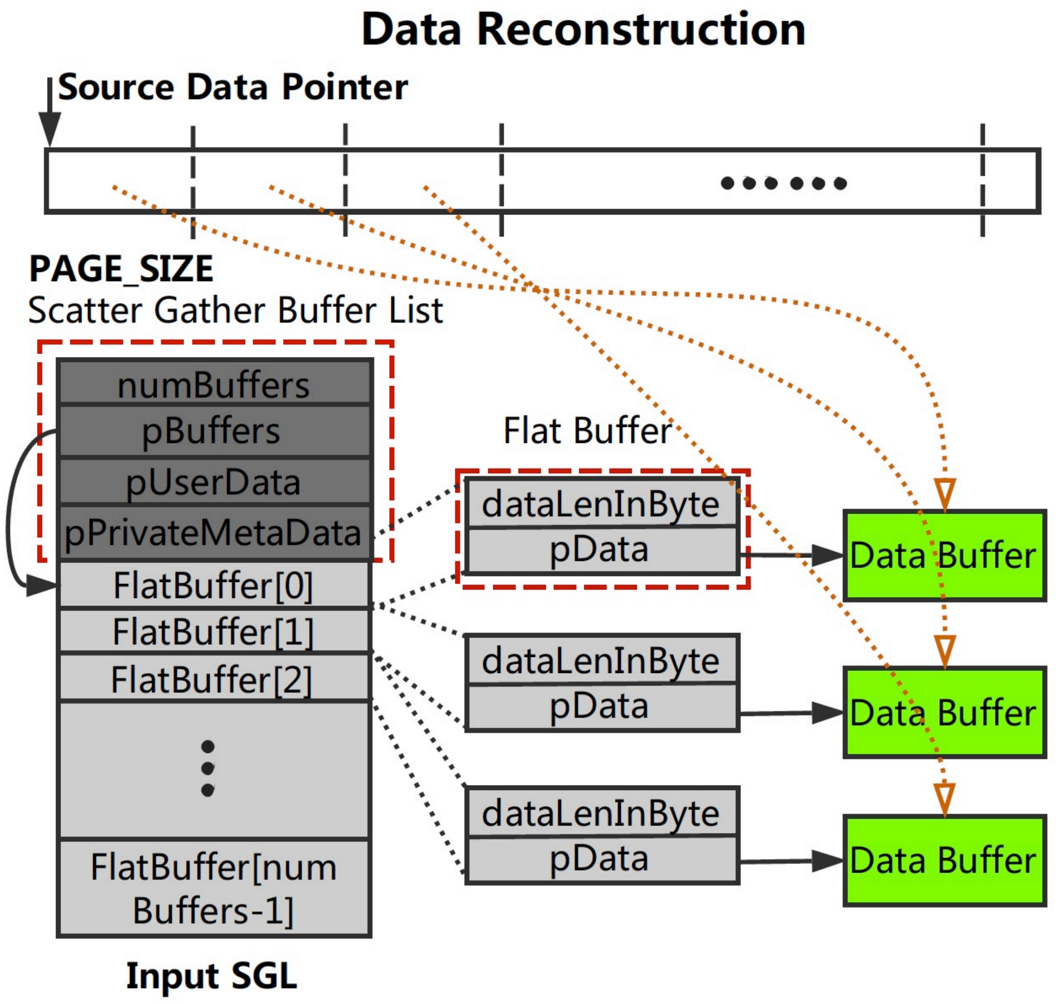
- **Algorithm selector**
  - QAT-accelerated gzip by default
  - Uniform interface (easily extended)
  - Availability: runtime error → switch to the software alternative

- **HW/SW switch** by source data size (4KB ~ 1MB)
  - < 4KB: benefits offset by offload overhead (QAT requests/responses, PCIe transactions, …)
  - > 1MB: large pre-allocated kernel memory as intermediate buffers
- **Compressibility-dependent** offloading (10% threshold for space saving)
  - Low compressibility means that data are not worth being stored in a compressed format
  - Original uncompressed data is returned

QAT Offloading Module

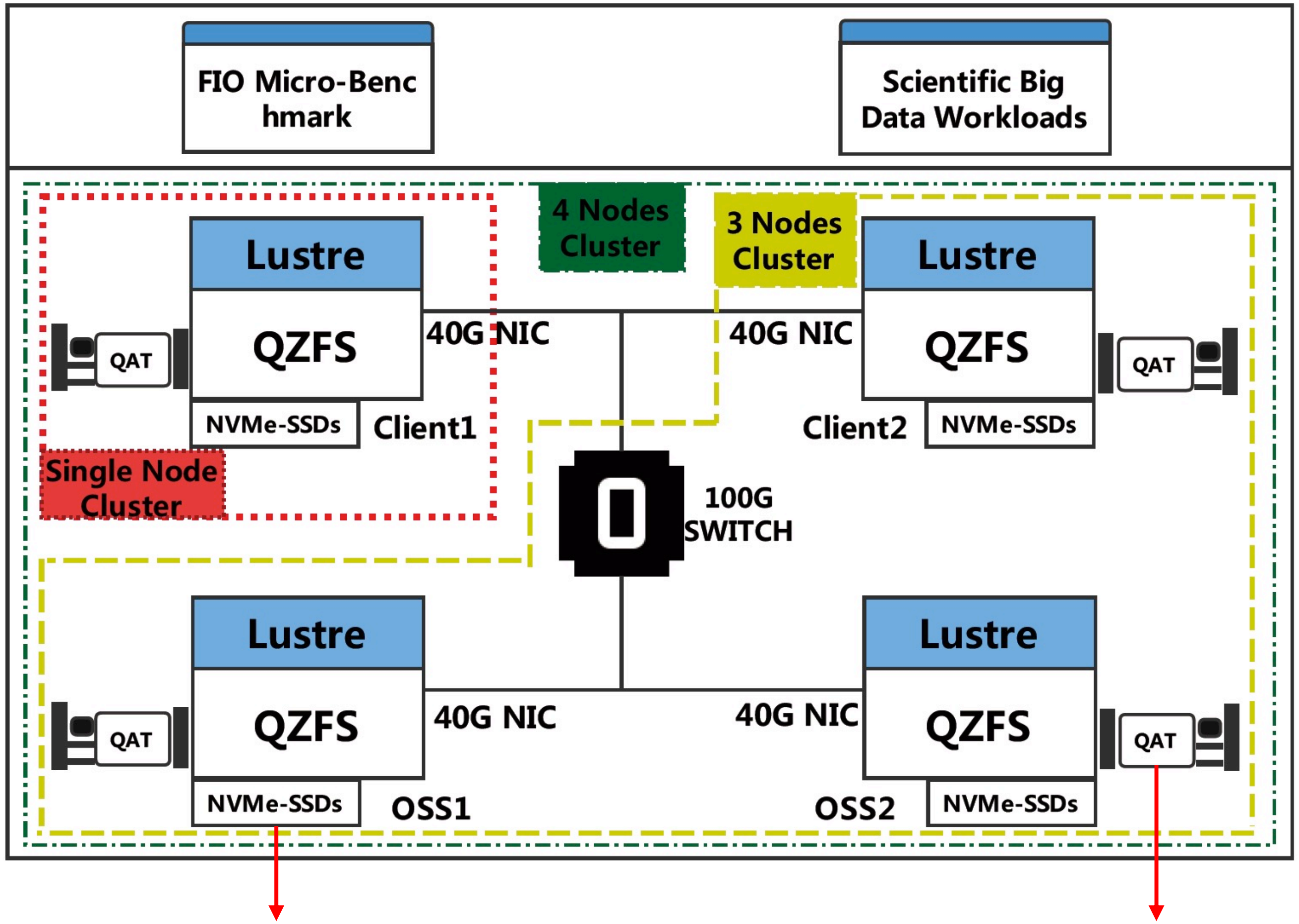
- Data prepared by ZIO uses virtual memory
- But QAT HW requires *contiguous physical memory* for DMA operations
- Data reconstruction: **zero memory copy**

- Vectored I/O : scatter/gather buffer list (SGL), partition by page frames
- *numBuffers* = S_{src} >> *PAGE_SHIFT* + 2
  zero buffer handled by QAT
- Differentiate *vmalloc* and *direct memory* region
- Physical page: *kmap* for long-lasting mapping



E.g., 11KB source data = 2KB + 4KB +4KB + 1KB

Evaluation



SSD Array: three 1.6TB Intel® P3700 NVMe SSDs

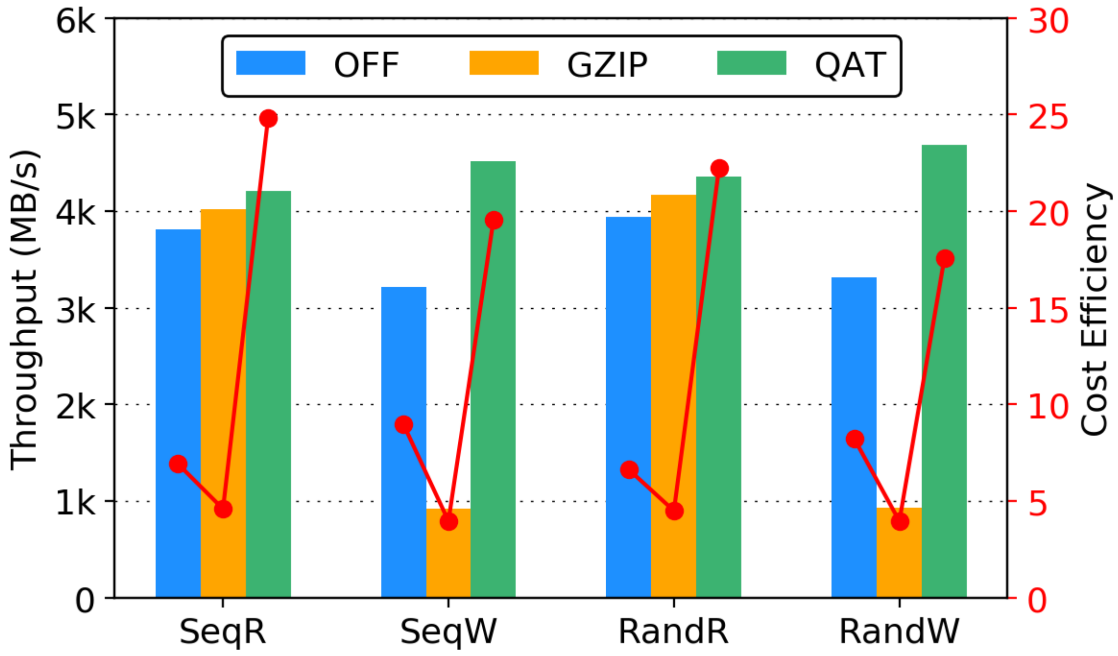
DH8950: 24Gbps

- Lustre cluster with varying nodes
- Four-node cluster: two clients and two OSSes (object storage servers)
- Two benchmarks
  - FIO micro-benchmark
  - Genomic data post-processing
- Cost-efficiency metric

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

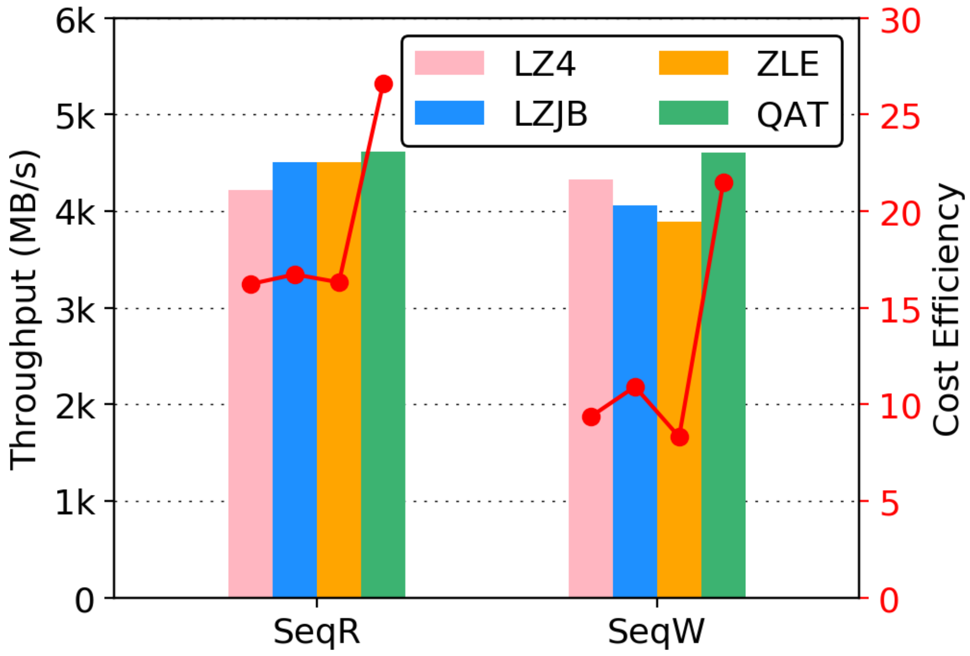
FIO Micro-benchmark

- 16 FIO threads in each client with fixed FIO block size



OFF	14%	11%	15%	12%
GZIP	82%	95%	84%	96%
QAT	15%	19%	16%	20%

compression ratio = 3.5 ~ 3.8

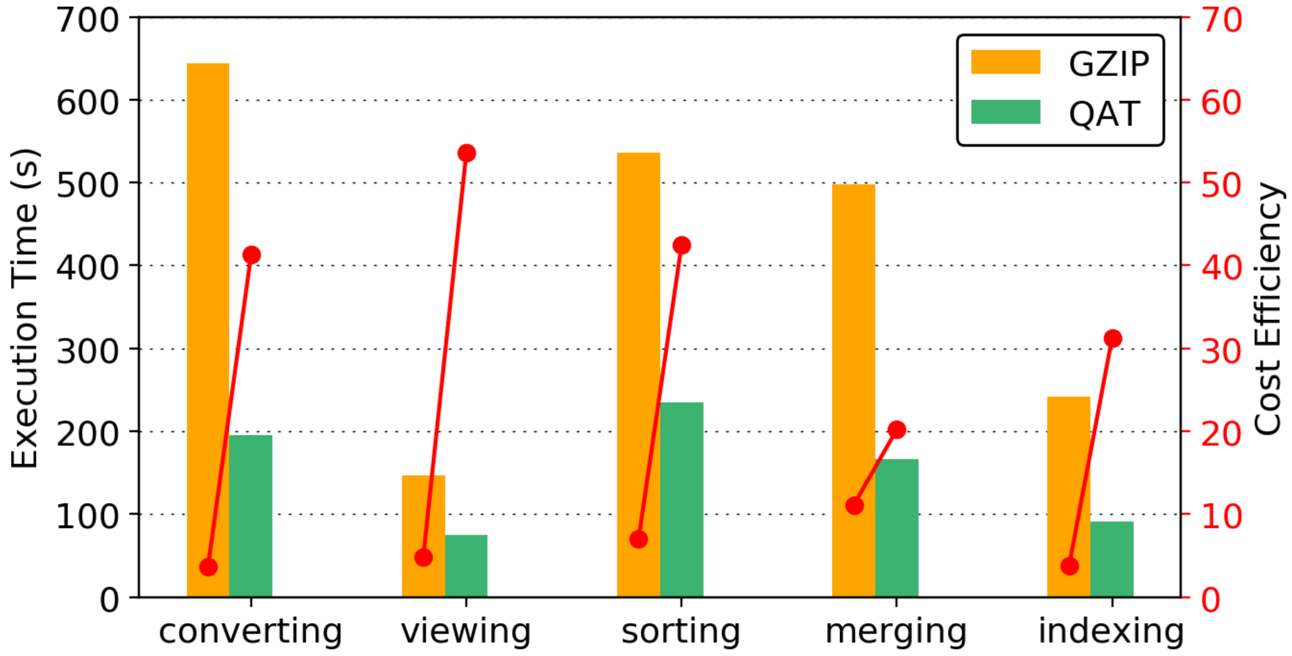


LZ4	2.0	13%	22%
LZJB	2.5	15%	23%
ZLE	1.6	10%	19%
QAT	3.7	14%	17%

Genomic Data Post-processing

- *SAMTools*, five operations involving read/write I/O
- 8 processes (multi-thread) in *one* client to manipulate 76GB genomic data

↓63%



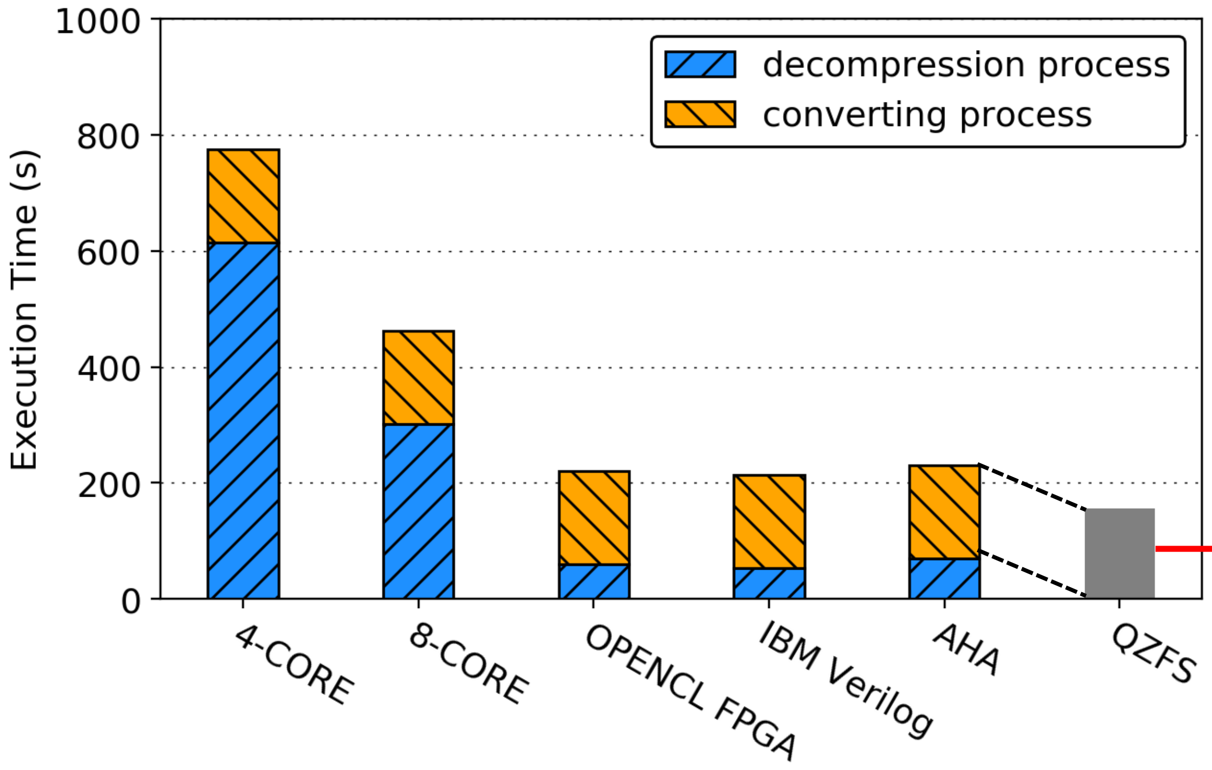
6X↑

GZIP	64%	47%	67%	51%	54%
QAT	7%	4%	8%	13%	11%

compression ratio = 3.4 ~ 4.2

Genomic Data Post-processing

- **QZFS vs. Simple gzip (application layer)**
  - decompression process: read compressed data & decompression & write uncompressed data
  - converting process: read uncompressed data & converting & write new format back



Application can achieve similar performance by integrating well-designed compression module: fragmentation/compression, multi-thread, QAT ...

But each new application may involve heavy modifications.

read/write compressed data

decompression & converting simultaneously

Bottleneck Analysis

- **FIO highest 4680 MB/s, not achieve hardware limit**
  - CPU% in two OSSes: 20.2%
  - SSD Array throughput: 4680/3.55 = 1318 MB/s < 3314 in OFF case
  - NIC (40GbE) throughput: 2340 MB/s = 18.72 Gbps
  - QAT throughput: 18.72 Gbps (80% of 24Gbps limit)
- **Bottleneck**
  - # of ZFS worker threads with offloading ability: limited by # of QAT instances
  - A worker thread interacts with QAT in synchronous mode: the next compression request cannot be submitted until the completion of the previous one.
  - More FIO threads cannot give rise to more parallel/concurrent QAT requests

Asynchronous Offload Mode

- **Overview**
  - One thread → one QAT instance
  - When: fully utilize QAT accelerator with limited # of threads
  - What: one thread can concurrently offload multiple compression tasks
- **Async implementation**
  - Async support in all layers of ZFS software stack to handle an uncompleted task
  - Efficient pause (context saving) and resumption (context restoring) of an offload job *in one worker thread*
  - Re-entering of a same handler: state flag, fiber/coroutine, …
  - Reference: QTLS, a high-performance SSL/TLS asynchronous offload framework (published in PPoPP ’19)

Q & A

Codes:

- QZFS (into ZFS Linux release): https://github.com/zfsonlinux/zfs
- Async Mode Nginx (QTLS): https://github.com/intel/asynch_mode_nginx
- QATzip Library (similar to zlib): https://github.com/intel/QATzip

Contacts: hxkcmp@sjtu.edu.cn, weigang.li@intel.com