applied sciences

MDPI

Article

QZRAM: A Transparent Kernel Memory Compression System Design for Memory-Intensive Applications with QAT Accelerator Integration

Chi Gao [1], Xiaofei Xu [2], Zhizou Yang [3], Liwei Lin [4,5] and Jian Li [3,*]

[1] Fifteen Department, Chengdu Aircraft Design Institute, Chengdu 610041, China
[2] Electronic Department, China Aeronautical Radio Electronic Research Institute, Shanghai 200233, China
[3] Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, Shanghai 200240, China
[4] Fujian Key Laboratory of Big Data Mining and Applications, Fuzhou 350118, China
[5] School of Computer Science and Mathematics, Fujian University of Technology, Fuzhou 350118, China
* Correspondence: li-jian@sjtu.edu.cn

Abstract: In recent decades, memory-intensive applications have experienced a boom, e.g., machine learning, natural language processing (NLP), and big data analytics. Such applications often experience out-of-memory (OOM) errors, which cause unexpected processes to exit without warning, resulting in negative effects on a system's performance and stability. To mitigate OOM errors, many operating systems implement memory compression (e.g., Linux's ZRAM) to provide flexible and larger memory space. However, these schemes incur two problems: (1) high-compression algorithms consume significant CPU resources, which inevitably degrades application performance; and (2) compromised compression algorithms with low latency and low compression ratios result in insignificant increases in memory space. In this paper, we propose QZRAM, which achieves a high-compression-ratio algorithm without high computing consumption through the integration of QAT (an ASIC accelerator) into ZRAM. To enhance hardware and software collaboration, a page-based parallel write module is introduced to serve as a more efficient request processing flow. More importantly, a QAT offloading module is introduced to asynchronously offload compression to the QAT accelerator, reducing CPU computing resource consumption and addressing two challenges: long CPU idle time and low usage of the QAT unit. The comprehensive evaluation validates that QZRAM can reduce CPU resources by up to 49.2% for the FIO micro-benchmark, increase memory space (1.66×) compared to ZRAM, and alleviate the memory overflow phenomenon of the Redis benchmark.

Keywords: memory compression; memory-intensive application; NLP; OOM; QAT

Citation: Gao, C.; Xu, X.; Yang, Z.; Lin, L.; Li, J. QZRAM: A Transparent Kernel Memory Compression System Design for Memory-Intensive Applications with QAT Accelerator Integration. *Appl. Sci.* **2023**, *13*, 10526. https://doi.org/10.3390/app131810526

Academic Editor: Vincent A. Cicirello

Received: 25 April 2023
Revised: 2 September 2023
Accepted: 5 September 2023
Published: 21 September 2023

1. Introduction

With the development of hardware, the number of CPUs and GPUs is increasing, and computing capacity has been greatly enhanced. This indicates that high-performance computers and large-scale data applications can increasingly promote scientific research breakthroughs. This further increases the demand for memory capacity in computers. Memory-intensive applications are also growing rapidly, for example, natural language processing (NLP) [1,2], fluid dynamics in climate simulations, particle simulations in astrophysics, quantum mechanics, and so on. Obviously, such applications require more and more memory. Due to the continuous growth of memory requirements, applications often experience out-of-memory (OOM) errors.

An OOM error is a common type of error for programmers. It is one of the most important bottlenecks in memory-intensive applications and can seriously affect and even kill applications. As an example, in NLP, a pre-training algorithm can train models through a series of universal languages to obtain a large number of datasets required for proprietary

tasks. Google engineers have found that the addition of the Lamb optimizer to BERT [3] training can greatly shorten training time by increasing the batch size. However, this can lead to a sharp increase in BERT's demand for memory space, which can easily lead to a system memory overflow. When increasing the batch size to 1600 on ×86 servers, the 240 G memory space will be quickly exhausted, and the BERT training system will eventually interrupt the training task due to OOM errors. However, simply increasing costly physical memory cannot solve the ever-increasing memory demands [4]. Memory is the resource that restricts the development of such applications. So, efficient application storage and processing of data will be an integral part of future storage systems. Data compression re-encodes data using specific compression algorithms to reduce the storage space occupied by the data, thereby improving data transmission, storage efficiency, and processing efficiency [5,6]. So, memory compression, as one of the measures used to handle increasing memory requirements, has been proposed and has rapidly developed.

Major operating systems have introduced swap-space compression (e.g., Linux's ZRAM [7]) as a type of memory compression. The main idea of these schemes is to increase the number of available memory pages by compacting inactive pages. Any access to the compressed pages would trigger a page fault, prompting the OS to decompress the page into a full-page frame, which is more efficient than reading from secondary storage. However, these schemes introduce additional computing loads, affecting the overall performance of upper-layer applications.

This paper proposes a high-performance memory compression system known as QZRAM. QZRAM has efficient data compression and provides application-agnostic and cost-efficient data storage. It is based on the Intel® QAT accelerator [8]. Quick-Assist Technology (QAT) is a hardware acceleration technology launched by Intel for network security and data storage. ZRAM (mostly called compcache before 2014) is a very important function in Linux. It is mainly used to temporarily increase memory capacity by compressing cold pages. ZRAM compresses the inactive virtual pages and transfers them to physical memory to free virtual memory. If the compressed pages are accessed by applications again, ZRAM will decompress them and return them to virtual memory. Such a design can prevent cold pages from being exchanged with low-speed secondary hardware devices (such as SSD and HDD), which reduces the delay in accessing data and improves the performance of applications. Thanks to the high-performance computing of QAT, QZRAM can use the GZIP algorithm and offload it to simultaneously attain a high compression ratio and low computing resource consumption. In detail, QZRAM adds two new modules instead of the original (de)compression procedures to offload compression to QAT. Firstly, QZRAM adds a parallel page-based write module, which enables QZRAM to better utilize multiple cores of computers and multiple computing units of QAT to concurrently process write requests. Secondly, QZRAM adds a compression module based on a parallel compressed stream, provides a transparent compression function entry, and manages the QAT hardware-accelerated GZIP algorithm and pure software compression algorithm libraries.

In our evaluation, we use QZRAM as the swap device and measure its performance in the FIO micro-benchmark and the practical Redis benchmark. In the FIO micro-benchmark, QZRAM can attain a 1.6× average write throughput improvement with a 1.51× compression ratio and about a 43% reduction in computing resource consumption (from more than 49.2% CPU consumption to less than 36.6% CPU consumption). In the practical Redis benchmark, by leveraging QAT acceleration, QZRAM (1) increases memory space (1.66×) based on ZRAM, delaying the timing of OOM triggers; and (2) significantly reduces the consumption of CPU resources during compression so that the CPU resources can be better allocated, improving the performance of the upper-layer of the Redis server.

This paper makes the following contributions:

- We propose QZRAM, which leverages the Intel® QAT accelerator and integrates it into ZRAM. In QZRAM, this integration can achieve efficient data compression and provide application-agnostic and cost-efficient data storage. QZRAM is designed in

the kernel, making it transparent to upper-level applications. It can support various memory-intensive applications without modification.
- We implement a flexible and dynamic memory compression engine in QZRAM. Since the GZIP compression algorithm is assisted by QAT, QZRAM improves the data compression ratio based on ZRAM, thereby improving memory capacity and delaying memory overflow opportunities.
- We add an offload module to QZRAM, which asynchronously offloads (de)compression functions to the QAT accelerator. This way of offloading can reduce the CPU computing resource consumption during the QZRAM data compression process and address the technical challenges of long CPU idle times and low usage of QAT computing units.
- We evaluated the performance of QZRAM through experiments, and the results show that QZRAM can effectively reduce CPU resources in the FIO micro-benchmark, increase memory space based on ZRAM, and alleviate the memory overflow phenomenon of the Redis benchmark.

2. Background

In this section, we introduce the relevant technical background of memory compression, analyze ZRAM in detail, and state the motivation of hardware-assisted memory compression for memory-intensive applications.

2.1. OOM Errors in Memory-Intensive Applications

OOM errors are a common issue in databases, mobile devices, JVMs, deep learning, and more. With the increase in memory capacity demand, memory-intensive applications may easily suffer from OOM errors, which can seriously affect a system's performance. For example, in NLP training, a large amount of task-specific data is required, and manually labeled data are insufficient. In the pre-training phase, a series of training universal language models are developed to obtain the large number of datasets required for proprietary tasks. To reduce the training time, researchers add the Lamb optimizer to BERT training, which increases the batch size and significantly increases the memory overhead, potentially causing system OOM errors. But at the same time, it leads to a sharp increase in the cost of memory, so memory space can be quickly exhausted, and the BERT training system may eventually interrupt the training task. So, memory space has become one of the key resources of modern and future computer systems.

However, simply expanding physical memory capacity is not a good solution [4]. There are two main reasons for this: (1) DRAM occupies a significant portion of the costs of the entire computer system and power consumption; and (2) due to signal-integrity issues, today's high-frequency storage channels prohibit many DRAM modules from connecting to the same channel, thereby limiting the maximum number of DRAMs in the system. In contrast, memory compression has become an excellent solution that can increase the memory capacity of computer systems without incurring significant hardware/energy costs. Memory compression can increase the memory capacity of a computer system without increasing physical memory, thus meeting the needs of applications running larger data volumes for computation.

2.2. Memory Compression

Memory compression is widely used in servers, smartphones, and other computer systems to save memory space [9]. In a memory compression system, virtual memory is compressed and stored in physical memory or secondary storage devices (HDDs or SSDs). Compressed memory blocks are set to inaccessible states, subsequent access to the data triggers page faults and invokes the swap system to decompress and restore it. Compression is compute-intensive. Software memory compression schemes [7,10,11] are often bottlenecked by the computation overhead. Other memory compression schemes [9,12–18] are performed by a dedicated processor between the CPU cache and physical memory, which is called hardware memory compression.

2.3. ZRAM Memory Compression System

We revisit a popular Linux kernel software memory compression subsystem, namely the ZRAM module. As shown in Figure 1, ZRAM compresses the inactive virtual pages and transfers them to the physical memory, which can free a lot of physical memory. When the physical memory is almost used up, the ZRAM module transfers some compressed data to secondary storage until they are needed again.

ZRAM can virtualize one or more compressed RAM disks, which can be used as swap devices.

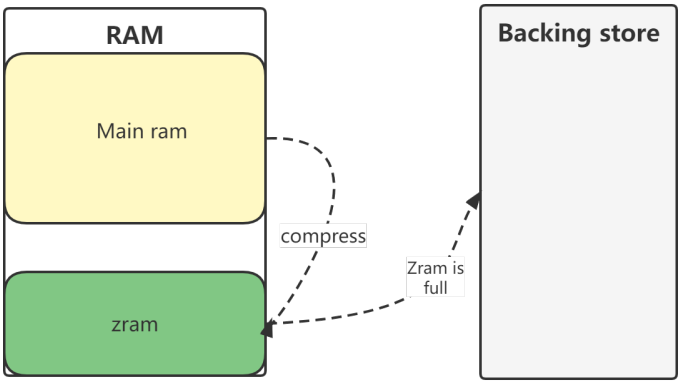


Figure 1. ZRAM module.

Processing flow: We depict the compression workflow to understand how ZRAM works and where ZRAM is located in Linux memory management, as shown in Figure 2. We divide the execution process of requests into three layers for clarity and intuition: the request process layer, the read/write page layer, and the compression library layer. When memory reclaims and swaps out inactive pages to ZRAM, the request process layer first iterates through the newly swapped-out request to parse it into a collection of pages and then transfers it to the read/write layer. Furthermore, the read/write layer calls the compression function of the compression library layer to compress the data and finally copies the compressed data to the zspages in Zpool [19], which is constructed of virtual devices. For swap-in requests, ZRAM calls the decompression function to decompress the compressed data and return them to the application layer. Note that the whole execution flow of ZRAM is a single-threaded operation. When the compression library layer calls algorithms implemented by software to (de)compress data, the entire execution process would be blocked until data (de)compression is complete. In other words, the next request can be processed only after the current request is completely processed.

Compression algorithms: Compression algorithms reduce the given storage space of data, but this also results in significant performance degradation. Furthermore, a high compression ratio generally means low-speed compression for compression algorithms.

We conducted an experiment to confirm the above and evaluate the algorithms to observe the reasons for ZRAM's bottlenecks. Our platform is equipped with Intel Xeon E5-2699 v4 processors, 64 G of RAM at 2133 MHz, and is running Linux kernel 4.8.12. ZRAM supports the LZO [20], LZ4 [21], and other compression algorithms [22–24]. This experiment begins with the FIO micro-benchmark with different compression algorithm configurations. Table 1 shows the throughputs, CPU usage, and compression ratios of the different compression algorithms.

Observation 1: From the experimental data, we can see that (1) The LZx algorithms have a low compression ratio, although they have a higher throughput. The purpose of memory compression is to increase memory capacity, so only a high enough compression ratio can make a significant difference. (2) The compression algorithms implemented in pure software consume a lot of CPU resources, which inevitably affects the performance of upper-layer applications.

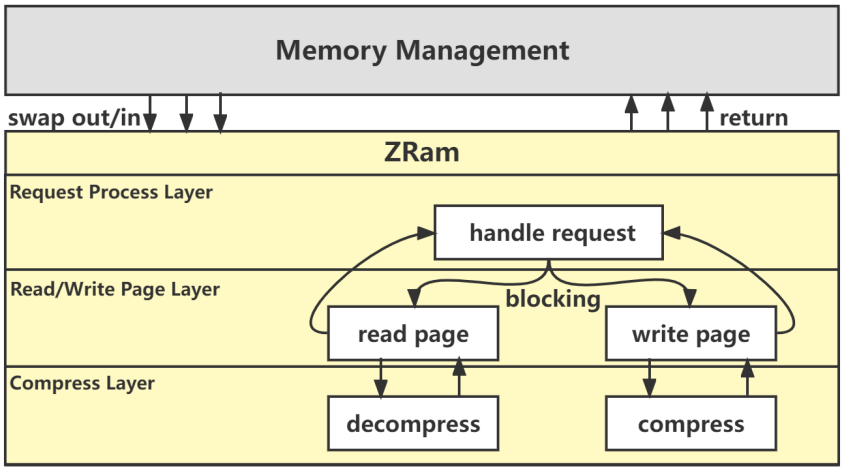


Figure 2. ZRAM processing flow.

Table 1. Compression algorithm performance test.

	Left: Read / Right: Write		
Mode	**MiB/s**	**CPU Usage [%]**	**Comp Ratio**
LZO	1588 / 670	60.20 / 55.10	1.49
LZ4	1973 / 723	78.57 / 86.36	1.51
DEFLATE	588 / 65.5	90.57 / 96.99	2.25

2.4. Hardware-Assisted Data Compression

In Section 2.2, we observed that the compression tasks not only consume a lot of costly CPU resources but are also repetitive. If ZRAM uses some dedicated hardware processors to handle these resource-intensive tasks, the performance of ZRAM may be improved. Fortunately, many hardware accelerators (e.g., FPGA [25], GPU [26], ASIC [27], etc.) have emerged to assist CPUs with all kinds of computation-intensive tasks. Intel Corporation has also launched its ASIC accelerator, QAT [8], which can help the CPU in compressing or encrypting data. Many researchers have proposed efficient heterogeneous systems [28,29]. However, there are practical challenges in heterogeneous systems, one of which is how to design a high-performance hardware and software collaboration scheme.

Direct offloading data compression to the QAT accelerator reduces CPU consumption during compression. However, the native offloading mode, as shown in Figure 3, faces some practical challenges. Although the QAT driver provides a non-blocking interface for I/O calls, synchronous I/O calls used by the compression module to submit offloading requests cannot be returned directly (i.e., block). Unfortunately, this frequent blocking may result in a waste of both CPU and QAT resources. Firstly, a large number of CPU cycles are used for waiting. After a compression request is submitted to the QAT hardware, QZRAM needs to wait for its response. Whether QZRAM adopts busy_loop or sleep for waiting, the I/O request processing will be paused for a period of time. Secondly, the parallel computing unit inside the QAT hardware has a low utilization rate. For each request process, only after the current I/O request is completed by the heterogeneous accelerator can the next compression request be submitted. In other words, only one accelerator unit can be used at the same time.

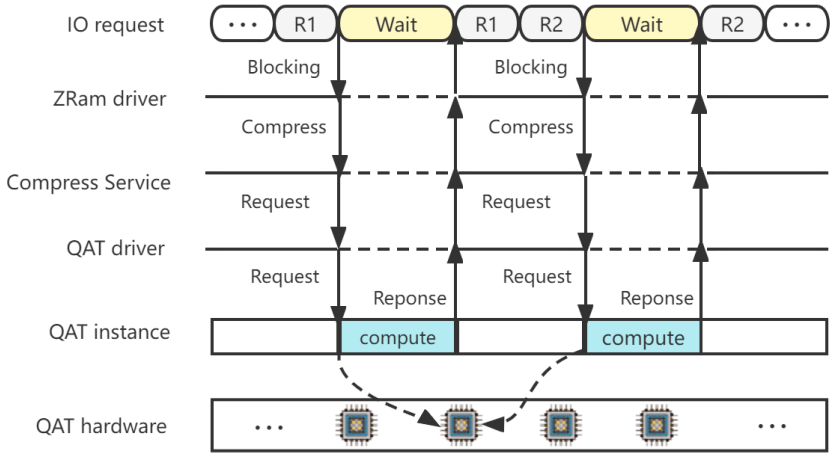


Figure 3. Direct offloading mode.

3. System Design

Above all, the ZRAM module can increase the memory capacity of computers, which delays the trigger time of OOM errors, but computation-intensive compression tasks inevitably degrade the performance of upper-layer workloads. Not only that, simply offloading the compression operator to the hardware accelerator results in some performance bottlenecks. Therefore, we propose a high-performance memory compression system QZRAM. The overall QZRAM architecture is illustrated in Figure 4, and the yellow parts are the newly added or modified modules relative to ZRAM. The overall QZRAM architecture includes memory-intensive applications (e.g., database, AI training, etc.), the Linux memory management system, and QZRAM, as well as the underlying physical memory and QAT accelerator.

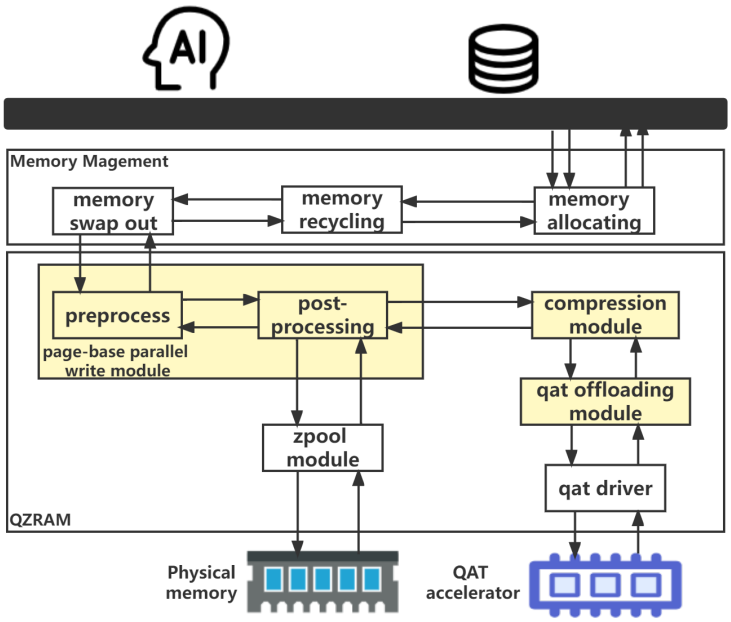


Figure 4. Overall architecture of QZRAM.

If memory capacity cannot meet the requirements of the upper-layer applications, memory reclaim will swap the inactive pages to QZRAM. First, the page-based parallel write module preprocesses write requests and sends them to workers for the second half of processing. Second, workers send the original data to the compression module for data compression. Finally, the compression module calls the hardware-assisted GZIP algorithm or software-implemented algorithms to compress the data and return them to the worker. If

the compression module selects the hardware-assisted GZIP algorithm, the QAT offloading module will offload the compression operator to the QAT accelerator.

3.1. Page-Based Parallel Write Module

Through careful analysis of ZRAM, we found that ZRAM cannot handle new requests asynchronously. In other words, a request cannot be processed until the previous request is complete. Moreover, the time-consuming part of the whole process is the data compression and write operations, which are less sensitive to latency than read operations. Therefore, we decouple the process flow and construct a page-based parallel write module, as shown in Figure 5. For a new request, the QZRAM main thread preprocesses it, encapsulates it into the data required by workers in the postprocessing, adds it to the page request queue (First In, First Out), and allocates a worker from the thread pool to postprocess the request. Later, the worker retrieves the request from the shared queue and continues the process. This means that workers perform the compression operation and return asynchronously after the completion of the entire procedure, notifying memory reclamation that the processing of the current page has been completed.

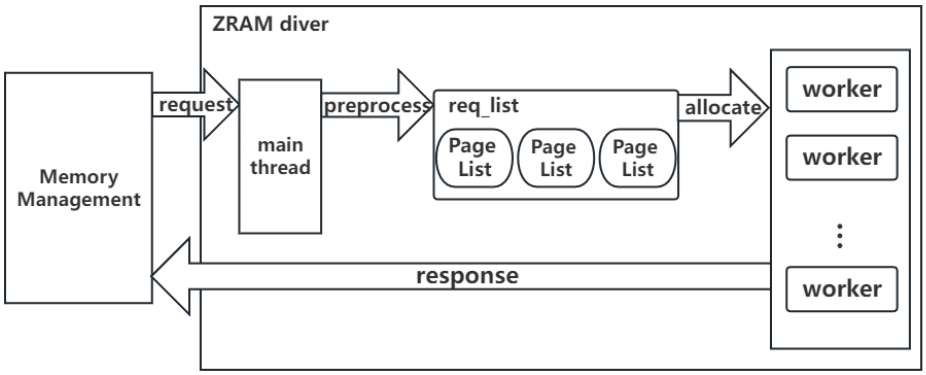


Figure 5. Page-based parallel write module.

Preprocessing: For a new request, the main thread of QZRAM checks the flag bit of the current request to determine whether it is a delete operation. The request is then traversed and parsed at the virtual page granularity and sliced into the page collection because QZRAM's processing granularity is exactly the page size such as 4 k (1024 bytes). For the pages in these page sets, the first QZRAM main thread calculates the virtual page's QZRAM in-device index, in-page offset, and data length; the second QZRAM main thread incorporates in-device indexes, in-page offsets, data lengths, and other data structures needed for postprocessing into one object of page_list. QZRAM main thread will then add page_req to page_list and continue parsing the remaining physical pages until all the physical pages for the current write request have been parsed. Finally, the QZRAM main thread adds page_list to the FIFO (First In, First Out) queue req_list and allocates worker threads from the thread pool to continue the postprocessing work.

Postprocessing: Once the workers begin running, they loop through the req_list queue to obtain the requests encapsulated by the main thread and then perform write operations for each page in the request. The worker obtains the workflow on the current CPU and uses the compression function of the compression module through the transparent compression entry provided by zstream (compression stream). After the zstream submits the compression request, it goes to sleep for a short time until the compression is complete or times out. Once the compression module returns, the worker immediately resumes and executes the rest of the processing flow. When the process is complete, the worker thread notifies the memory reclaim that the swap-out request processing for the current page has been completed. If the worker accesses the request queue and finds that there are no pending requests in the queue, the worker actively goes to sleep and releases the CPU, waiting for the main QZRAM to wake up and reenter the execution state.

3.2. Compression Module

The compression module provides a transparent compression function to the upper layer. The appropriate compression algorithm is selected to (de)compress the data according to the preset configuration of the user. The overall design of the compression module includes three parts: concurrent compression streams, the algorithm implemented in the software, and the GZIP library assisted by the QAT accelerator.

Concurrent compression stream. Compression streams provide a compression entry that supports concurrent read and write operations, enabling the worker thread to concurrently execute the second half of the process. In addition, the compression stream maintains a buffer to store the compressed data. During compression, there is a small probability of data inflation. Therefore, the compressed stream buffer space is set to 8k, twice the size of the virtual page. By default, the GZIP algorithm library based on the QAT accelerator is used as the compression algorithm in the compression stream. If users have special requirements, they can select a compression algorithm for the compression stream based on the application scenario.

3.3. QAT Offloading Module

To address the practical challenges of direct offloading, this section proposes an asynchronous offloading design, as shown in Figure 6. After the request is preprocessed by the QZRAM main thread, the worker thread continues with the time-consuming part of the process. Multiple worker threads simultaneously invoke unified compression algorithms of the compression module. Then, the compression module selects the submitted GZIP compression request to the QAT driver and sleeps until it receives the interrupt signal from the QAT driver. This approach not only avoids the CPU's idle wait time after submitting the compression request but also enables QZRAM to process multiple requests concurrently and multiple computing units of QAT hardware to work simultaneously. Finally, after processing the compression request, the QAT accelerator sends an interrupt signal to the CPU indicating that the compression is complete. Then, the QAT driver uses the callback function to wake up the worker, and it resumes the compression task and continues the remaining write process.

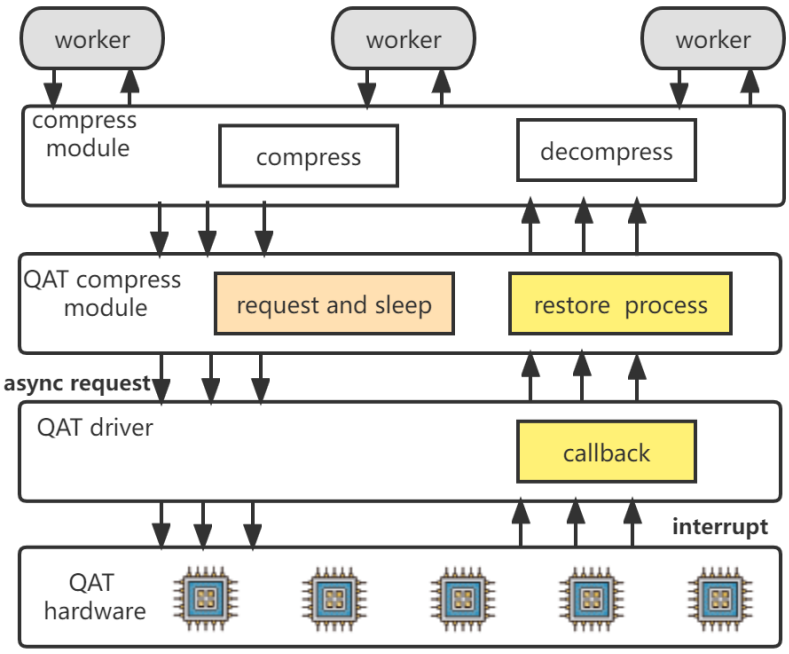


Figure 6. QAT asynchronous offloading design.

4. Implementation and Optimization

In this section, we elaborate on which modules QZRAM implements and optimizes based on ZRAM.

4.1. Compression Module

High reliability and high compatibility: To ensure the high reliability of QZRAM, the compression module introduces many reliable mechanisms and strategies for data compression. In the compression ratio detector, to prevent the memory overflow from directly causing runtime errors, which can crash the system, the compression module conducts a detailed boundary check for each physical page I/O operation and ensures its atomicity during data operation. In the management of the compression algorithm, if the hardware-assisted compression algorithm does not work, the compression module uses the corresponding algorithm implemented in pure software as a replacement to ensure that compression can be successfully completed. This mechanism makes QZRAM independent of the underlying hardware platform. Even if the QAT accelerator is damaged or there is no device, it can still complete the compression work, which is a highly reliable solution to meet the data compression scenario. In addition, the compression module is transparent to the upper-level module and has high compatibility with different compression algorithms.

4.2. QAT Offloading Module

4.2.1. Handling Buffer Overflow

When QZRAM is inserted into the kernel, the QAT offloading module initializes the compression buffer, compression instance, and compression session and binds them to the underlying QAT accelerator. The QAT offloading module offloads compression requests to the request ring through a compression instance and obtains results from the response ring after the data are compressed. However, in the process of compression, the data may swell because the effect of compression is not obvious, and the header and tail are added to the compressed data for the sake of compatibility with software and hardware memory algorithms. The data inflation generated in the process of data compression may still have a serious impact. The compression buffer is a continuous physical memory area, if overflow occurs in it, QZRAM may have serious errors, resulting in an entire system crash or even physical damage to the QAT hardware. Therefore, to prevent buffer overflow, the QAT offloading module sets the buffer size to twice the original data. Such a design may result in a certain amount of memory wastage, but it is worth it compared to the potentially serious consequences of a buffer overflow.

4.2.2. Load Balance and Availability

Load balance: In scenarios of high-performance computing, upper-layer applications have large-scale data, resulting in extremely high compression bandwidth. If the computing resources of the QAT accelerator are exhausted, the QAT instances cannot be obtained by the QAT offloading module. To ensure the full utilization of computing resources in the system, the load balancing between QAT hardware-assisted compression and traditional compression performed by CPUs is required in the QAT offloading module. Therefore, if the QAT instances in the QAT offloading module are insufficient, the QAT offloading module transfers all compression operations to the software compression algorithms processed by the CPU. When the computing resources of the QAT accelerator are replenished, compression resumes using the QAT accelerator, achieving a seamless transition between the two schemes.

High availability: The QAT session is a service abstraction that describes the configuration information and parameters of the compression service in the QAT instances. In the requests and responses of the QAT instances, specific data manipulation is responsible for the QAT session, and each session represents a (de)compression service. After the QAT offloading module completes instance initialization and data reconstruction, the QAT session is responsible for the information transfer to the QAT accelerator. Thus, if a

software-level error occurs during compression, such as passing the wrong DMA (Direct Memory Access) address or the wrong number of parameters, it is handled and recovered in the QAT session. If a hardware-level error occurs, the QAT offloading module clears all instances and sessions, reclaims the memory of all the corresponding data structures, sets the available identifier location of this module to false, and then restarts the QAT driver and the QAT offloading module. Finally, all relevant buffer data structures, logical instances, and compression sessions are re-initialized for use in the next phase of continued work. As a mature commercial ASIC device, the QAT accelerator device rarely experiences errors at the hardware level. However, the implementation of a perfect error-handling mechanism in the QAT offload module can help avoid potential risks and improve system availability and robustness.

5. Evaluation

In this section, we introduce the experimental platform and evaluation of QZRAM. Additionally, we use the traditional file read/write micro-benchmark tool FIO (Flexible I/O) and the distributed memory database Redis [30]. The former is used to test various performance improvements of QZRAM compared to ZRAM. The latter shows whether QZRAM can result in performance improvements in memory-intensive applications in real big-data application scenarios.

5.1. Evaluation Methodology

Experimental testbed: We establish an experimental testbed with three physical servers, each of which is equipped with two 88-core Intel Xeon E5-2699 v4 processors, 64 G RAM, and one Intel C62x PCIe QAT card. Each server runs on CentOS 7 (Linux kernel 4.8.12) and is equipped with a QAT 1.7 hardware driver for Linux. Finally, we connect these servers via an Intel 40 GbE NIC.

Experimental dataset: The Calgary dataset [31] was created at the University of Calgary. Since then, the Calgary dataset has gradually become most widely used in the field of compression, especially text compression. To evaluate the compression system practically, it consists of nine different types of text data, including typical English writing (e.g., bibliography and news), computer programs, transcripts of terminal sessions, and so on.

Performance metrics: We mainly compared four configurations: no-swap, ZRAM (LZO), ZRAM-QAT (QAT direct offloading mode), and QZRAM. No-swap disables the swap partition, which is the foundation of this evaluation. ZRAM means that the OS uses ZRAM as the swap device, which configures the LZO algorithm to compress data. ZRAM-QAT means that the operating system selects ZRAM as the switch device and changes the compression algorithm to the QAT hardware-assisted GZIP algorithm. QZRAM represents the modification of the single-threaded write process to a multi-threaded write process in the asynchronous offloading mode.

5.2. Evaluation Benchmark

Micro-benchmark workloads: FIO (flexible I/O tester) is a micro-benchmark tool that tests device storage performance by simulating practical read/write behavior. The principle of FIO is that it generates a buffer of text data and then uses multiple threads to concurrently write the data to the specified device to test the performance of the device. Note that all packets received by ZRAM are virtual pages with a fixed size of 4 k. So, the packet block size is specified as 4 k by default in the experiments unless otherwise specified.

Database workloads: Redis (remote dictionary server) [30] is an in-memory database widely used in the industry. It is mainly used as a high-performance cache server and can also be used as a MOM (Message-Oriented Middleware) or in session sharing. In addition, using Redis, it is easy to simulate the production of large amounts of data. Therefore, Redis is used to simulate real memory-intensive scenarios to test the improvement in the performance of upper-layer applications. The Redis experiments are performed as follows:

1. The Redis server starts up as many server instances as there are CPUs and then runs each server instance exclusively on a dedicated CPU.
2. All server instances are initialized. Redis is configured not to use persistent storage, and instances are initially set to "empty".
3. Once all server instances are started, all Redis clients use the Calgary dataset to generate values and send SET requests concurrently to instances on the server.
4. Until all SET operations have been processed by the server, the client starts to GET previous values from the server concurrently.
5. The above 3–4 steps are repeated three times, and the master client records the detailed experimental data.

5.3. FIO Micro-Benchmark

Compression ratio and CPU usage. The compression ratio represents the size that a specific storage space can store the original data, which reflects the memory space improvement, and the CPU usage displays the CPU resource consumption during data compression. The compression ratio is mainly counted and recorded in the log file by the statistics module of the ZRAM (Linux kernel). The experimental results are shown in Table 2. We can see that the compression ratio of QZRAM with the GZIP algorithm configuration is 1.51 times that of the default ZRAM, and the CPU usage of QZRAM is 36.60% lower than that of ZRAM. Thanks to the powerful parallel computing capability of the QAT accelerator, QZRAM not only has a higher compression ratio compared to ZRAM but also greatly reduces the consumption of CPU resources.

Table 2. Compression ratio and CPU usage test.

	CPU Usage (%)	Compression Ratio
Linux kernel (ZRAM)	55.10	1.49
QZRAM	18.50	2.25

Write throughput and latency. Since the processing granularity of the QZRAM memory block device is a page size (4 k), the block size in this experiment was set to gradually increase from 4 k to 4 m. Figure 7 shows the write throughput of ZRAM, ZRAM-QAT, and QZRAM under different block-size configurations. Although the previous experiment demonstrated that replacing the LZO algorithm implemented only in software with the GZIP algorithm assisted by the QAT hardware accelerator can indeed improve the data compression ratio and reduce the consumption of CPU computing resources during the compression process, this experiment shows that directly unloading the compression operator onto the QAT hardware accelerator does not improve the processing performance of the ZRAM memory compression system. The write throughput of ZRAM-QAT (QAT direct unloading mode) is even less than 1/10 of that of ZRAM, and the I/O processing latency is still 10 times that of ZRAM. The main reason for these results is the low utilization rate of a large number of CPU cycles used for waiting and QAT parallel computing units. Therefore, this article adds page-based parallel read and write modules and a compression operator asynchronous unloading scheme to QZRAM. In the graph, it can be seen that the write throughput of QZRAM with the new parallel read and write modules and asynchronous offloading is higher than that of ZRAM under different block sizes, especially when the block size is 32 k, and QZRAM's throughput is 1.71 times that of ZRAM. The experiment has proven that the asynchronous unloading compression operator designed in this article addresses the technical challenge of the direct unloading mode. It also proves that QZRAM has a stronger write operation I/O processing ability compared to ZRAM in both large and small packet conditions.

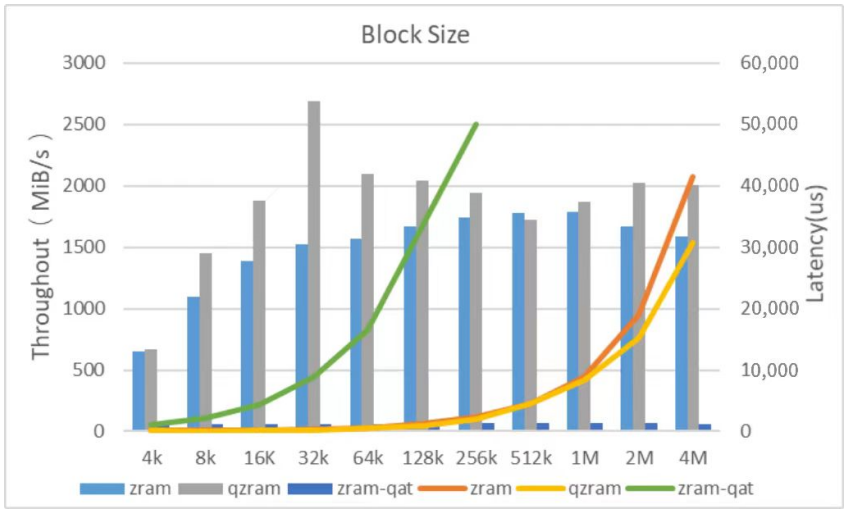


Figure 7. Write throughput and latency with different block sizes.

I/O pattern. Swap partitions send both sequential-swap (i.e., sequential I/O) and single-swap (i.e., random I/O) requests to swap devices. Figure 8 shows the throughput and CPU usage of the experimental subjects in four I/O modes. In sequential read, the ZRAM throughput is 1588 MiB/s and CPU usage is 60.10%. Relatively speaking, the CPU usage of ZRAM-QAT (i.e., direct offloading mode) is reduced to 16.79%, but the throughput also drops to 105MiB/s due to the large number of idle CPUs in direct offloading and the low usage of the QAT acceleration unit. As a result, QZRAM introduced asynchronous offloading, increasing throughput to 611MiB/s and maintaining CPU usage at the low level of 21.60%. In sequential write, QZRAM has roughly the same throughput as ZRAM, and QZRAM has a 36.7% reduction in CPU consumption compared to ZRAM. On the whole, this experiment proves that QZRAM effectively tackles the challenges existing in the direct offloading design and greatly releases CPU resources while maintaining performance.

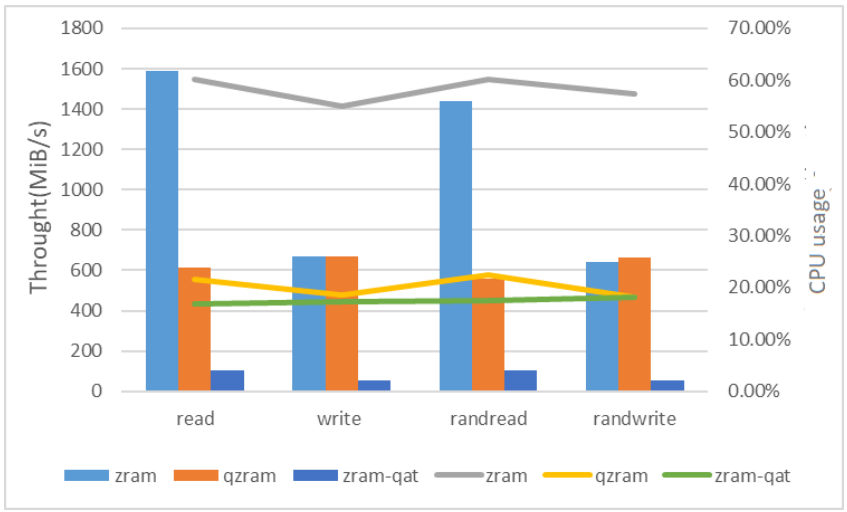


Figure 8. Performance of different IO patterns.

Scalability. Swap devices may receive requests from multiple processes. Therefore, QZRAM needs a strong ability to handle requests in parallel. Figure 9 shows the performance of QZRAM and ZRAM under different workloads. When the workload is 1, although the throughput of both ZRAM and QZRAM is around 670MiB/s, the CPU usage of QZRAM is 36.6% lower than that of ZRAM. Both the throughput and CPU usage of ZRAM and QZRAM increase as the workload increases. With a workload of 16, the QZRAM CPU usage is reduced by up to 49.2% compared to ZRAM. This means that

QZRAM saves a lot of CPU resources without decreasing the throughput. At the same time, we can see that the throughput of either ZRAM or QZRAM does not increase with the increase in the workload because it increases the competition for resources within the system. In addition to resource competition, we suspect that the QZRAM throughput did not increase due to the limited number of compression instances per QAT accelerator card (18 compression instances), which is obviously not enough to be used by 88 workers at the same time. One solution is to insert more QAT accelerator cards into the machine because if the physical machine has a large number of cores, using a few more QAT cards can maximize the potential of QZRAM. In other words, QZRAM has a stronger parallel processing ability compared to ZRAM. Figure 10 shows the read and write request latency of ZRAM and QZRAM under different workloads. In the graph, it can be seen that the average read delay of QZRAM increases by 54 microseconds compared to ZRAM, which is consistent with the throughput difference between the two in the second experiment. At the same time, the write delay of QZRAM is essentially the same as that of ZRAM. This result is partly due to the additional delay introduced by the PCIE data transmission between the QAT and CPU in the compression process. On the other hand, it is because the GZIP algorithm implemented by the software in the QAT hardware uninstallation module adds a header and footer to the compressed data packets to be compatible with the compressed data.

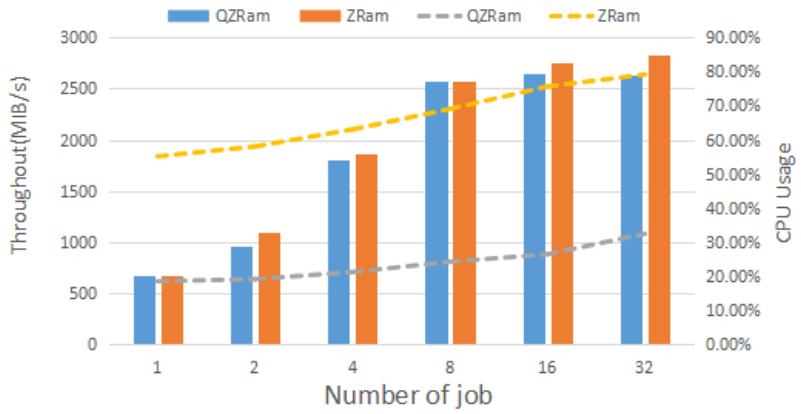


Figure 9. Performance of different workloads.

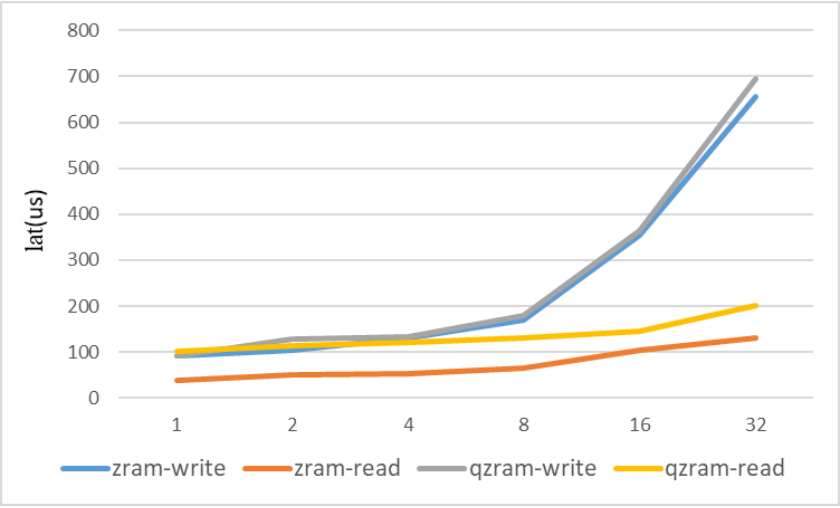
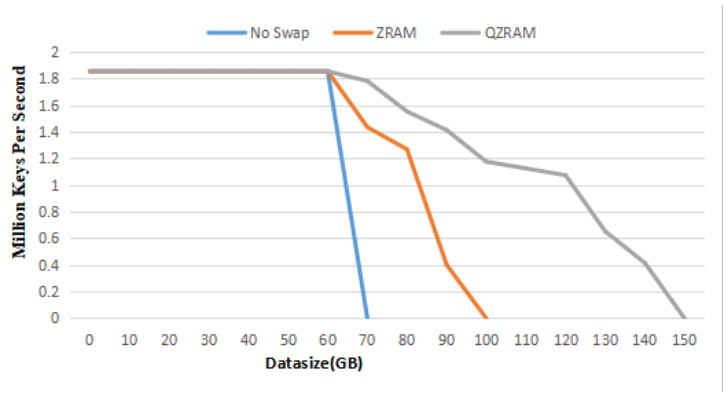


Figure 10. Performance of different workloads.

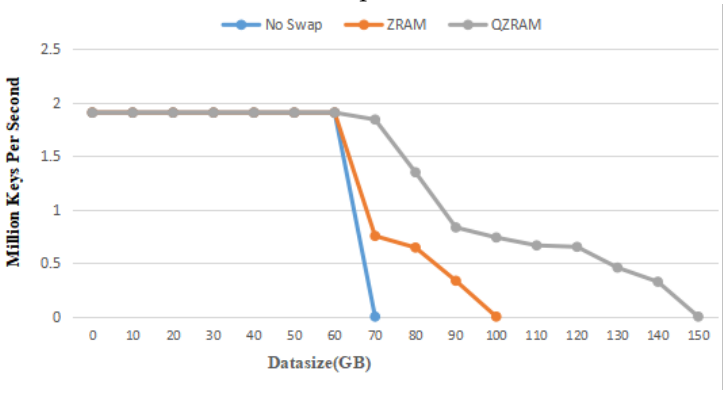
5.4. Redis Evaluation

Memory compression is mainly used in memory-intensive applications (e.g., databases). Therefore, we use the Redis benchmark to determine the extent to which QZRAM improves the performance of memory-intensive applications.

SET operation: Figure 11a shows the throughput of the three subjects. We can see that the curves of the three configurations are almost even (around 1.8 million keys/s) before 60 G because they have enough memory capacity and do not need to swap out inactive pages. When the data size reaches 80 G, the remaining memory space of the no-swap devices (i.e., non-swap-equipped devices) is insufficient for the Redis servers to run, so the OOM program selects and kills the Redis servers that occupy a large amount of memory space. On the contrary, the others retain a certain amount of memory space by swapping out inactive data to the swap partition. Although the life cycles of the other subjects are extended, their performance drops, particularly ZRAM. In detail, the throughput of ZRAM is only 68% of that of the no-swap configuration. The relative QZRAM throughput is 83% of the throughput of the no-swap configuration. This is because the pure software compression algorithm used in ZRAM consumes a large amount of CPU resources, and the single-threaded process of ZRAM is overwhelmed when dealing with large-scale data. In contrast, QZRAM maintains good performance when the OS performs memory recycling, and the throughput of QZRAM only decreases by 17% compared to that of the no-swap devices. As the data size continues to increase, the performance of ZRAM decreases more and more due to the high concurrency pressure and the low compression ratio of LZO. In the test where the data size is 90 G, the memory space is completely exhausted, and the Redis server is forcibly killed by the OOM program. In contrast, the throughput of QZRAM decreases slowly with the increase in the working set, and the OOM phenomenon does not appear until 150 G due to the fact that QZRAM further increases the memory capacity. In conclusion, QZRAM has better I/O performance and 1.66 times more memory space than ZRAM.



(**a**) SET operation



(**b**) GET operation

Figure 11. Performance in Redis benchmark testing.

GET operation: After the SET requests are all processed, the Redis clients concurrently send GET requests to the servers to obtain the values of the SET requests. Figure 11b shows the throughput of the Redis clients with the increase in the working set in the three configurations. We can see that the three throughput curves are almost even (1.91 million keys/s) before 60 G. This is because at this time, the Redis data of the three are stored in the physical memory (i.e., not swapped out to swap devices). When the data size exceeds 70 G, the throughput of both ZRAM and QZRAM degrades. Compared to the no-swap configuration, the throughput of ZRAM decreases by more than 60%, and that of QZRAM only decreases by around 10%. These experimental results are generated because the software-implemented compression algorithm used in ZRAM consumes a large amount of CPU resources, and the single-threaded blocking read process of ZRAM is overwhelmed when dealing with the huge amount of data in the highly concurrent scenario, which affects the processing performance of the upper layer of the Redis server. In addition, the reason the GET operations suffer more significant performance degradation compared to the SET operations is that the GET requests are issued after all the SET requests have been completed. If the Redis clients SET more than 60 G data to the server, the GET operation will occur after the swap partition is used. In this case, when the Redis server accesses the data, it is likely to trigger a page fault, reminding the OS to decompress the data from the swap partition and return it to memory. And because the memory is insufficient, the OS might swap out and swap in pages at the same time. In general, QZRAM has better read performance compared to ZRAM under high-load memory compression scenarios.

6. Conclusions

We analyzed two problems of ZRAM and discussed the practical challenge of hardware-assisted data compression for memory-intensive applications. QZRAM offloads GZIP compression to the QAT accelerator to reduce CPU resources and increase memory space based on ZRAM. In addition, we proposed an asynchronous offloading scheme to not only avoid CPUs' idle wait time after submitting the compression request but also enable QZRAM to submit multiple compression requests concurrently. QZRAM is designed at the kernel level, making it transparent to upper-level memory-intensive applications. The evaluation showed that QZRAM can reduce CPU resources by up to 49.2% in the FIO micro-benchmark, increase memory space (1.66×) based on ZRAM, and alleviate the memory overflow phenomenon of the Redis benchmark.

Author Contributions: Conceptualization, C.G., X.X. and Z.Y.; Methodology, J.L.; Validation, C.G., X.X., Z.Y.; Resources, X.X.; Writing—original draft, Z.Y. and L.L.; Writing—review & editing, C.G.; Supervision, J.L. All authors have read and agreed to the published version of the manuscript.

Funding: This paper was supported in part by NSFC grant number 61972245.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Liu, P.; Yuan, W.; Fu, J.; Jiang, Z.; Hayashi, H.; Neubig, G. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.* **2023**, *55*, 1–35. [CrossRef]
2. Choi, H.; Kim, J.; Joe, S.; Gwon, Y. Evaluation of bert and albert sentence embedding performance on downstream nlp tasks. In Proceedings of the 25th International Conference on Pattern Recognition (ICPR), Milan, Italy, 10–15 January 2021; pp. 5482–5487.
3. Siddiq, M.L.; Santos, J.C. Bert-based github issue report classification. In Proceedings of the 2022 IEEE/ACM 1st International Workshop on Natural Language-Based Software Engineering (NLBSE), Pittsburgh, PA, USA, 8 May 2022; pp. 33–36.
4. Mittal, S.; Vetter, J.S. A survey of architectural approaches for data compression in cache and main memory systems. *IEEE Trans. Parallel Distrib. Syst.* **2015**, *27*, 1524–1536. [CrossRef]
5. Ziv, J.; Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* **1977**, *23*, 337–343. [CrossRef]

6. Burrows, M.; Wheeler, D. A block-sorting lossless data compression algorithm. In Proceedings of the Digital SRC Research Report Citeseer, Snowbird, UT, USA, 25–27 March 1994.
7. Gupta, N. ZRAM Project. *Linux Foundation*; Linux Corporation: San Francisco, CA, USA, 2020.
8. Intel. Intel® QAT: Performance, Scale and Efficiency. 2022. Available online: https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html (accessed on 6 June 2023).
9. Panwar, G.; Laghari, M.; Bears, D.; Liu, Y.; Jearls, C.; Choukse, E.; Cameron, K.W.; Butt, A.R.; Jian, X. Translation-optimized Memory Compression for Capacity. In Proceedings of the 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO), Chicago, IL, USA, 1–5 October 2022; pp. 992–1011.
10. Jennings, S. The Zswap Compressed Swap Cache. 2013. Available online: https://lwn.net/ (accessed on 1 June 2023).
11. Sanchez, D.; Kozyrakis, C. The ZCache: Decoupling ways and associativity. In Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Atlanta, GA, USA, 4–8 December 2010; pp. 187–198.
12. Park, S.; Kang, I.; Moon, Y.; Ahn, J.H.; Suh, G.E. BCD deduplication: Effective memory compression using partial cache-line deduplication. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Las Vegas, NV, USA, 12–14 December 2021; pp. 52–64.
13. Tremaine, R.B.; Franaszek, P.A.; Robinson, J.T.; Schulz, C.O.; Smith, T.B.; Wazlowski, M.E.; Bland, P.M. IBM memory expansion technology (MXT). *IBM J. Res. Dev.* **2001**, *45*, 271–285. [CrossRef]
14. Ekman, M.; Stenstrom, P. A robust main-memory compression scheme. In Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05), Madison, WI, USA, 4–8 June 2005; pp. 74–85.
15. Pekhimenko, G.; Seshadri, V.; Kim, Y.; Xin, H.; Mutlu, O.; Gibbons, P.B.; Kozuch, M.A.; Mowry, T.C. Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, Minneapolis, MN, USA, 19–23 September 2013; pp. 172–184.
16. Zhao, J.; Li, S.; Chang, J.; Byrne, J.L.; Ramirez, L.L.; Lim, K.; Xie, Y.; Faraboschi, P. Buri: Scaling big-memory computing with hardware-based memory expansion. *ACM Trans. Archit. Code Optim.* **2015**, *12*, 1–24. [CrossRef]
17. Kim, S.; Lee, S.; Kim, T.; Huh, J. Transparent dual memory compression architecture. In Proceedings of the 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), Portland, OR, USA, 9–13 September 2017; pp. 206–218.
18. Qian, C.; Huang, L.; Yu, Q.; Wang, Z.; Childers, B. CMH: Compression management for improving capacity in the hybrid memory cube. In Proceedings of the 15th ACM International Conference on Computing Frontiers, Snowbird, UT, USA, 26–28 March 2018; pp. 121–128.
19. ZPool. 2009. Available online: https://www.kernel.org/doc/html/v5.0/vm/zsmalloc.html (accessed on 6 June 2023).
20. Oberhumer, M. LZO (Lempel-Ziv-Oberhumer) Data Compression Library. 2013. Available online: http://www.oberhumer.com/opensource/lzo/ (accessed on 1 June 2023).
21. Collet, Y. LZ4 Compression Algorithm. 2020. Available online: https://github.com/lz4/lz4 (accessed on 6 June 2023).
22. Collet, Y.; Turner, C. Smaller and Faster Data Compression with Z Standard. Facebook Code, Volume 1. 2016. Available online: https://engineering.fb.com/2016/08/31/core-data/smaller-and-faster-data-compression-with-zstandard/ (accessed on 1 June 2023).
23. Plauth, M.; Polze, A. Towards Improving Data Transfer Efficiency for Accelerators Using Hardware Compression. In Proceedings of the 2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW), Pangkal, Indonesia, 23–24 October 2018.
24. Harnik, D.; Khaitzin, E.; Sotnikov, D.; Taharlev, S. A fast implementation of deflate. In Proceedings of the 2014 Data Compression Conference, Snowbird, UT, USA, 4–7 April 2014; pp. 223–232.
25. Kuon, I.; Tessier, R.; Rose, J. FPGA architecture: Survey and challenges.*Found. Trends Electron. Des. Autom.* **2008**, *2*, 135–253. [CrossRef]
26. Owens, J.D.; Houston, M.; Luebke, D.; Green, S.; Stone, J.E.; Phillips, J.C. GPU computing. *Proc. IEEE* **2008**, *96*, 879–899. [CrossRef]
27. Smith, M.J.S. *Application-Specific Integrated Circuits*; Addison-Wesley: Boston, MA, USA, 1997; Volume 7.
28. Lee, S.; Park, J.; Fleming, K.; Kim, J.; Huang, L.; Yu, Q. Improving performance and lifetime of solid-state drives using hardware-accelerated compression. *IEEE Trans. Consum. Electron.* **2011**, *57*, 1732–1739. [CrossRef]
29. Hu, X.; Wang, F.; Li, W.; Li, J.; Guan, H. {QZFS}:{QAT} Accelerated Compression in File System for Application Agnostic and Cost Efficient Data Storage. In Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 19), Snowbird, UT, USA, 30 March–1 April 2019; pp. 163–176.
30. Eddelbuettel, D. A Brief Introduction to Redis. *arXiv* **2022**, arXiv:2203.06559.
31. Abel, D.I.J. Calgary Corpus. 2020. Available online: http://www.data-compression.info/Corpora/CalgaryCorpus/ (accessed on 6 June 2023).