Characterization of Data Compression in Datacenters

Geonhwa Jeong*
Georgia Institute of Technology
Atlanta, USA
geonhwa.jeong@gatech.edu

Bikash Sharma
Meta
Menlo Park, USA
bsharma@meta.com

Nick Terrell
Meta
Menlo Park, USA
terrelln@meta.edu

Abhishek Dhanotia
Meta
Menlo Park, USA
abhishekd@meta.com

Zhiwei Zhao
Meta
Menlo Park, USA
zhiweiz@meta.com

Niket Agarwal*
NVIDIA
Los Gatos, USA
agarwal.niket@gmail.com

Arun Kejariwal
Meta
Menlo Park, USA
akejariwal@meta.com

Tushar Krishna
Georgia Institute of Technology
Atlanta, USA
tushar@ece.gatech.edu

***Abstract*—Data compression has emerged as a promising technique to alleviate the memory, storage, and network cost with some associated compute overheads in warehouse-scale datacenter services. Despite being one of the most important components of the overall datacenter taxes, there has not been a comprehensive characterization of compression usage in datacenter workloads. Such characterization is paramount for both compression software developers and hardware accelerator designers as it can help them make optimal design trade-offs decisions in terms of performance, power, and cost while meeting service-level agreements of target applications. Moreover, it can provide data-driven insights to application developers to find optimal compression configuration choices for their services. In this paper, we first provide a holistic characterization of compression as used by various warehouse-scale datacenter services at a global social media provider, Meta. Next, we deep dive into a few representative use cases of compression in the production environment and characterize compression usage of the services while running live traffic. Finally, we conduct sensitivity studies to understand how different compression configurations are relevant to the overall infrastructure cost, followed by future research directions for compression hardware and software development.**

I. Introduction

Datacenter workloads are composed of different services which are often tightly coupled together with remote procedure calls (RPCs) [33], [41]. Numerous datacenter services such as advertisement predictions, caching systems, key-value stores, data warehouse, and machine learning platforms work in tandem to serve user requests [20], [25], [29], [33], [36], [39], [40]. In these services, data compression is widely used to reduce the amount of storage required for each service, the traffic resulting from the communication among services, and the memory total cost of ownership (TCO) by proactively compressing cold memory pages [31], [35]. In the current era of data deluge, data compression has become even more important than before for warehouse-scale datacenter services to efficiently manage unprecedented amounts of data, and bridge the growing gap between processing speed and I/O [18]. However, data compression comes at a computational cost, namely compute overhead required to perform the compression functions, adding to the *datacenter tax* [33], [40].

The focus of this paper is on characterizing both the data compression usage and its impact on overall infrastructure cost at Meta which runs one of the largest social media platforms. We believe this is the first work to perform such a characterization study at scale across hundreds of thousands of servers running datacenter applications/services. Specifically, at Meta, data compression is used to reduce the network costs for the communication of objects between different services, such as latency-sensitive Ads services [29], [40]. Data compression is also used for reducing storage costs of data that is stored across caching tiers and warm/cold storage nodes [20], [25], [36], [39]. For example, the data warehouse services apply compression to reduce the size of the data to efficiently utilize the finite amount of storage considering both retention time and compute/storage cost trade-offs.

There are three important **compression metrics** to evaluate the performance of compression: *compression ratio, compression speed, and decompression speed*. Compression ratio is measured as the original data size divided by the compressed size (the higher, the better). Compression and decompression speeds are the measures of how quickly the data can be compressed/decompressed by the system (the higher, the better). Improving compression ratio is crucial to enhance compression performance, which could significantly reduce the associated network/storage footprint cost. Meanwhile, improving compression and decompression speed can reduce the compute cost by reducing the time taken for compression and decompression. A recent work [12] has shown that 2× to 4× reduction in storage or network bandwidth utilization using compression can save tens to hundreds of thousands of US dollars per petabyte in storage and network cost.

To optimize the compute resource usage for compression and decompression, researchers and engineers have proposed several approaches based on software and hardware. On the software side, various compression algorithms including LZ4 [6], Zlib [8], and Zstandard (Zstd) [9] have been developed to improve compression performance. Generally, different compression algorithms provide different degrees of flexi-

*This work was done while Geonhwa Jeong was an intern and Niket Agarwal was an employee at Meta.

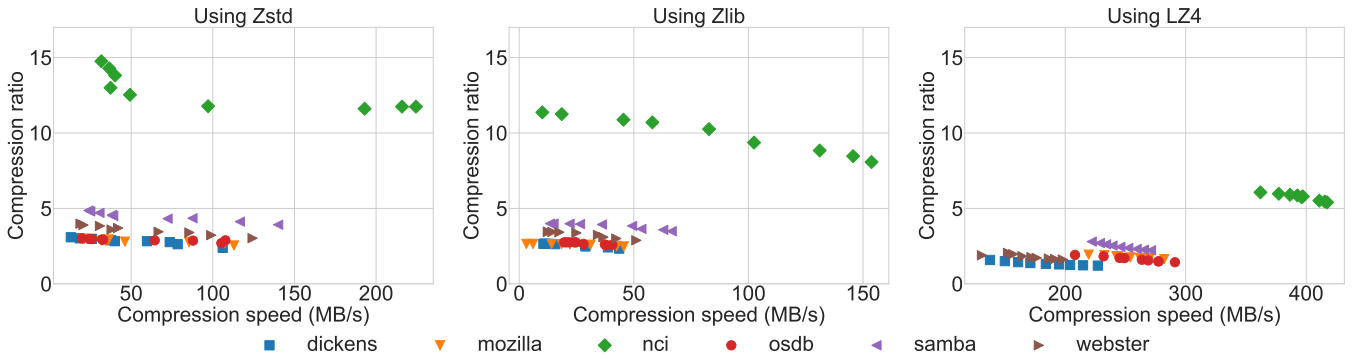


Fig. 1. Compression ratio and speed with different types of files in Silesia corpus using Zstd, Zlib, and LZ4.

bility through *compression levels*. Compression algorithms can use a larger match window to provide a higher compression ratio while sacrificing compression speed due to the overhead caused by the higher number of pattern matchings or vice versa. The users of these compression algorithms can tune the parameters such as the match window size indirectly by changing the compression level based on their application requirements. For instance, Zstd provides compression levels from -5 to 22, while Zlib offers ten compression levels from 0 to 9 to cover different use cases. Usually, higher compression levels take more computational resources to deliver higher compression ratios. To help the automation of the tuning process, services like Managed Compression [27] expose a stateless interface to users while the service keeps the states to train dictionaries using previous samples to provide a better performance, which we describe more in Section II-B. On the hardware side, both industry (Intel [11], [32], IBM [12], Oracle [15], and Microsoft [10], [26]) and academia [13], [21], [37], [44] have been actively working on developing on-chip and off-chip hardware accelerators to improve performance of compression.

In Figure 1, we compare three different compression algorithms (Zstd, Zlib, and LZ4) with different compression levels (from 1 to 9) to show the variation in compression performance due to differences in characteristics of the data being compressed. We use files from an excerpt of a well-known compression dataset, Silesia corpus [23], for this comparison. As the plots show, compression metrics depend heavily on the data, showing an order of magnitude difference in compression ratios and speeds. Hence, in addition to compression levels and algorithms, **it is important to understand the data characteristics of warehouse-scale applications** to design suitable compression algorithms and hardware accelerators for datacenter services. Surprisingly, to the best of our knowledge, there is no such characterization available for large-scale applications, so people have been relying on decades-old data sets such as Calgary corpus [19], Canterbury corpus [17], and Silesia corpus [23] when making design decisions for SW/HW based compression optimization [22], [26], [28], [34].

For orchestrating common datacenter operations, datacenter services spend huge compute cycles (*datacenter tax* [33]), which could be as high as 60% on some workload types [33], [40]. We measured the datacenter tax for compression across hundreds of thousands of servers in Meta and observed it to be 4.6% compute cycles on average. Some services spend as high as 30% of their compute cycles solely in compression, which corresponds to significant infrastructure costs.

The variance in compression metrics and costs discussed above also introduces a challenge to the teams maintaining and developing datacenter services: *How to find the best compression configuration for a service given the trade-offs between various metrics such as compression ratios, speeds, cycles overhead, and latency requirements?* The service characteristics in warehouse-scale workloads evolve quickly over time, making it prohibitively expensive to manually tune compression configuration every time the change lands in the fleet. Therefore, it is important to have a unified framework to estimate costs considering service-specific characteristics and requirements. To this end, we develop *CompOpt*, a first-order compression optimizer framework that quantifies the costs of integrating compression and associated system design choices, and we conduct a few sensitivity studies using this framework.

Summary of Contributions:

- We provide a fleet-level characterization of data compression usage across the global server fleet supporting one of the largest social media platforms, Meta, for different categories of datacenter services (Section III).
- We conduct a service-level characterization using a few representative services from different categories along with the explanation of various compression usages to illustrate different requirements and trade-offs for compression optimization (Section IV).
- We propose *CompOpt*, a first-order compression optimizer that leverages analytical models for estimating compression costs under the application constraints, and we conduct sensitivity studies using *CompOpt* to understand the impact of compression configurations to the overall cost on production applications (Section V).
- We propose future research directions for SW/HW compression acceleration based on our characterization and case studies on datacenter applications (Section VI).

II. Background

A. Compression in warehouse-scale datacenter services

Warehouse-scale datacenter workloads generally follow a service-oriented architecture approach to split the work required to process a query into smaller fractions that can be handled independently by different services. These services then follow an RPC-based approach to interact with each other and process the request before sending a response back to the user. Each of these individual services can scale to tens of thousands of servers in warehouse-scale systems. This service-based approach enables scaling applications for billions of users but can lead to significant processing overheads as services communicate with each other. Previous works [33], [40] have shown that these overheads at datacenters, known as *datacenter taxes*, can account for up to 40-60% of total compute cycles spent in the datacenters, motivating optimization to minimize these common overheads. Additionally, these works have shown that compression forms a significant portion of the overall datacenter tax to cut down the storage footprint and network traffic. Unlike the previous works that illustrate the datacenter taxes in general, we focus on **compression/decompression** and provide a detailed analysis of the specific use cases at Meta to explore opportunities for optimization and understand trade-offs clearly. We explain the details of the service categories and specific services in Section III and Section IV, respectively.

B. Compression algorithms

Services in the datacenter fleet of Meta use several different compression algorithms. The most prevalent (in terms of usage) algorithm for the services in the datacenter fleet of Meta is Zstd [9], [27], followed by LZ4 [6], and Zlib [8]. The three algorithms can be divided into two categories, LZ (Lempel-Ziv [45]) compressors (including Zstd and LZ4) and non-LZ compressors (including Zlib). We observed that more than 90% of compression used in the datacenter fleet at Meta is using LZ compressors, while Zlib is mostly used for backward compatibility with systems and applications that do not support Zstd/LZ4. Therefore, in the following, we focus on LZ compressors and describe their details.

LZ compressors. LZ compressors such as Zstd or LZ4 work by running a Lempel-Ziv [45] match-finding stage that finds repetition in the data stream and emits literals (bytes with no match) and sequences (offsets, literal lengths, and match lengths). The match-finding stage is followed by an encoding stage that takes the results of the LZ match-finding stage and emits the compressed data. LZ4 [6] is a simple and fast encoder that emits uncompressed literals. Then, it emits the sequence using byte-aligned variable-length integers. Zstd [9], [27] provides a better compression ratio than LZ4 by Huffman coding compressing the literals and compressing the sequences with Finite State Entropy [5]. This entropy encoding scheme improves the compression ratio but hurts decompression speed, as the decoder now has extra work to reconstruct the original data.

Trade-offs. Generally, compressors offer a trade-off between compression speed, compression ratio, and decompression speed. The trade-off between compression speed and compression ratio is dominated by the *LZ match-finding stage*; different match-finding algorithms can be used to find matches in the stage. The match-finding algorithms use heuristics and optimizations to find the best matches to succinctly represent the data in a given time budget. They range from fast greedy algorithms to slow dynamic programming algorithms which attempt to find the optimal encoding. Usually, the match-finding algorithm is selected by the compression level to trade off compression speed and compression ratio.

The trade-off between compression ratio and decompression speed largely depends on the *entropy encoding stage*; one can choose a fast but inefficient scheme like LZ4, or a slower but more efficient scheme like Zstd. Thus, decompression-latency-sensitive services might prefer LZ4 compression due to its high decompression speed, while storage-capacity-sensitive services might choose Zstd to reduce the size of the original data as much as possible.

Additionally, Zstd offers another optimization called dictionary compression to improve the compression of small data. LZ compressors typically struggle to compress small data because usually it does not contain many repetitions compared to large data. However, when compressing certain data (such as messages of the same type), there can be a lot of repetitions between them. LZ dictionaries are constructed ahead of time from sample data and capture these inter-message repetitions. Next, they are communicated out-of-band to the compressor/decompressor and used as shared history. Dictionary compression can be supported by a service architecture like Managed Compression [9].

C. Compression accelerators

Hardware acceleration for compression [10], [12], [32] is emerging as a promising way to provide an order of magnitude improvement in compression speeds and help free up compute cycles for other application work. Various compression accelerators are being actively developed both from industry and academia to alleviate the burden of compute units. For example, Intel has recently added Intel QuickAssist Technology (QAT) [32] which provides acceleration of compression as well as encryption and authentication. A previous work has integrated QAT with the ZFS file system to offload compression at the file system level to QAT [31]. IBM POWER and z15 integrated on-chip compression accelerators to reduce processor cycles and achieve memory/storage/network benefits [12] while only taking less than 0.5% of chip area for the integration of accelerators. Also, since they integrated the compression accelerators on-chip, they also save IO bandwidth/cost, which would have been required if ASIC or FPGA-based accelerators were used. Microsoft has introduced Project Zipline including RTL designs for compression and Project Corsica, which includes ASIC implementations of Zipline technology [10].

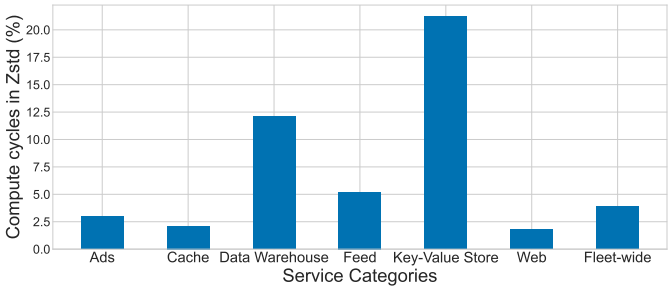


Fig. 2. Compute cycles (%) used by Zstd for different service categories.

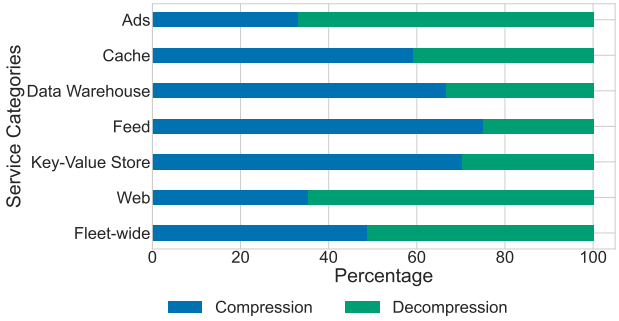


Fig. 3. Zstd compression and decompression split by compute cycles.

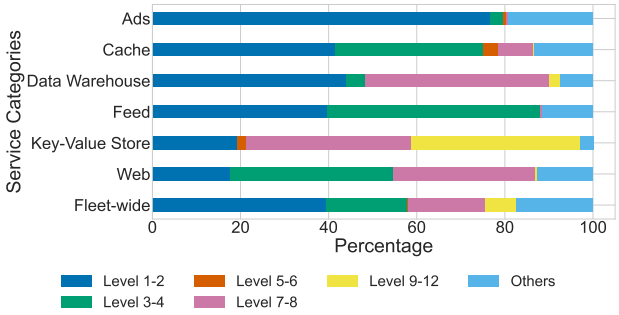


Fig. 4. Zstd levels usage by compute cycles.

III. Fleet-level Characterization

In this section, we illustrate the overall view of compression and decompression with holistic characterization across the Meta production services.

A. Characterization methodology

Diverse services can be categorized into a set of workloads that may share common patterns. To explore the suite of representative classes of workloads in Meta, we categorize some of the services in the datacenter fleet as follows: ***Ads, Cache, Data Warehouse, Feed, Key-Value Store, and Web***. To understand different dimensions of compression for the services, we conduct characterization leveraging the profiling infrastructure at Meta running continuously over a span of 30 days [14]. We look at sampled application call stacks in the profiling result, filter the call stacks for compression APIs, and aggregate cycles spent in relevant compression function calls including Zstd, Zlib, and LZ4.

B. Compression algorithms

To analyze the overall compression usage in the datacenter fleet, we measure the total compute cycles percentage with different compression algorithms, Zstd, Zlib, and LZ4. Across all services running in the datacenter fleet, we observe that an average of **4.6% of compute cycles** are spent for compression and decompression operations. This compression footprint as part of overall datacenter taxes is quite similar to the data reported in the previous datacenter characterizations [33], [40]. *Among the three compression algorithms, Zstd is dominant with 3.9% compute cycles while 0.4% and 0.3% are used for LZ4 and Zlib respectively. Thus, we focus on Zstd in this work for further detailed analysis to explain the finer aspects of compression, but the high-level insights could be shared with other compression algorithms as well since they share similar patterns to achieve the same goal.*

C. Compute usage by service categories

In Figure 2, we show the compute cycles percentage spent in Zstd for different service categories mentioned in Section III-A. We observe that there is considerable variance in compression CPU cycles percentage from 1.8 to 21.2% depending on service categories. We will illustrate comprehensive service-level characterization using a few services belonging to the categories in Section IV. *From the trend, we could identify that service categories directly related to managing the vast amount of data such as Data Warehouse or Key-Value Store spend a significant portion of compute cycles for compression and decompression.*

D. Compression/decompression split

In Figure 3, we show compression-decompression split for different service categories as well as the split for the entire fleet. Not surprisingly, we observe the different patterns for different service categories based on their use cases. We will discuss more details about service-level use cases in Section IV. Considering that decompression speed for a request is significantly faster (from 3× to 100×) than compression speed of the request, the plot indicates that the number of decompression calls is substantially higher than the number of compression calls across services. *Generally, this is due to the trend that various services involve writing compressed data to storage, such as a cache or database, and then reading the data from the storage multiple times (while there are some specific services where the number of compression calls and the number of decompression calls are nearly equal).*

E. Compression level usage

In Figure 4, we show how different Zstd compression levels are being used at the datacenter fleet. We identify that service owners tend to favor lower compression levels (1-4), which could imply that they prefer fast compression over a high compression ratio. For example, compression levels 1-4 take more than 50% of entire CPU cycles, which can be even

TABLE I
A SUMMARY OF TARGET DATACENTER SERVICES AT META

Service	Service Category	Description	Resource Boundedness	Key Takeaway
DW1	Data warehouse	Distributed data delivery service	Storage bound	Compute-storage cost trade-offs
DW2	Data warehouse	Distributed data shuffle service	Storage bound	Compute-storage cost trade-offs
DW3	Data warehouse	Distributed scheduling framework for data warehouse jobs	Storage bound	Compute-storage cost trade-offs
DW4	Data warehouse	Distributed scheduling framework for machine learning jobs	Storage bound	Compute-storage cost trade-offs
ADS1	Ads	Ads serving machine learning inference service	Network bound	Network compression and model variance
CACHE1	Caching	Distributed memory object caching service	Compute/memory bound	Small data compression
CACHE2	Caching	Distributed social graph data store service	Compute/memory bound	Small data compression
KVSTORE1	Key-value store	Large distributed key-value store	Storage bound	Different block sizes

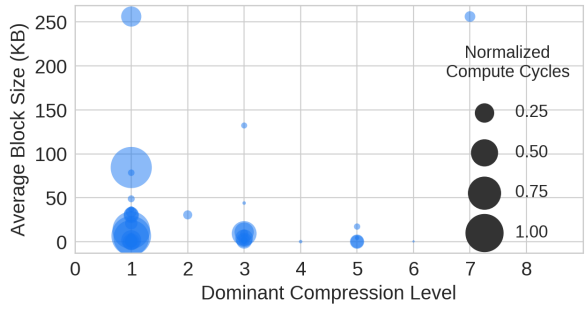


Fig. 5. Block size distribution across different services.

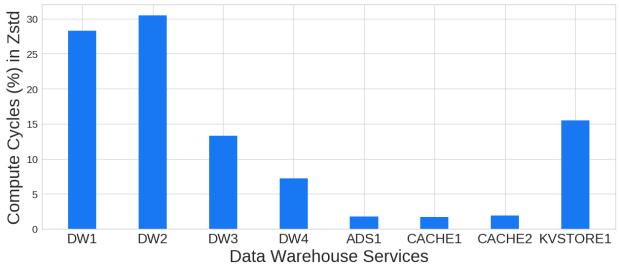


Fig. 6. Compute cycles (%) used by Zstd for different services.

higher than 80% for the Feed services. It is also possible that they have simply set the compression levels without thoroughly evaluating the storage-speed trade-offs. *Similar to the case of compression-decompression split, considering that compression speed with low compression levels is much faster than that of high levels, the number of services using low compression levels is significantly higher than the ones using higher levels.*

F. Compression data block sizes

Different use cases of compression pass vastly different data sizes to the compressor. Figure 5 shows the distribution of average data size across a set of services. Usually, larger inputs tend to compress better since there is not enough data for repetition in small inputs. However, compressed data does not offer random access; to read any byte of the data, the reader must decompress the entire blob. *Therefore, use cases that require random access must tune the block size to trade off compression ratio vs. decompression latency.*

G. Summary

In this section, we showed how compression is being used at the datacenter services in fleet-level and service category-level. The characterization shows the wide variety of compression usage in datacenter services in terms of 1) compute cycles spent by compression and decompression, 2) compression-decompression split, 3) compression levels, and 4) input block sizes. This characterization also indicates that we cannot easily generalize the usage of compression in datacenters (i.e. an optimization for a certain aspect of a compression use case does not guarantee its applicability to other use cases due to the diversity), and this motivates us for service-level characterization and optimization for compression.

IV. SERVICE-LEVEL CHARACTERIZATION

In this section, we characterize some of the largest users of compression cycles, Data Warehouse services (DW1-4), an Ads service (ADS1), caching services (CACHE1-2), and a Key-value store service (KVSTORE1) as summarized in Table I.

A. Service-level compute cycles used by compression

Figure 6 shows compute cycles spent by Zstd for different services. We observed that the compute cycles spent for compression vary dramatically across eight services that we explored (from 1.7% to 30.5%). We have also noticed that the sum of compute cycles spent in Zstd of the top-5 services is almost 50% of those spent in Zstd for thousands of services in the datacenter fleet.

B. Data warehouse services

Data Warehouse is a service that provides warm storage used for analytic workloads and cold storage for backups. It stores data in a columnar format called Optimized Row Columnar (ORC). Columns get encoded by the storage engine and then passed to Zstd in blocks of up to 256KB. Nearly all compression usage in Data Warehouse services is driven by reading and writing ORC files. While data is passed to Zstd in up to 256KB blocks, several notable workflows exist. **Ingestion** (DW1) reads the data from the data source, decompresses it (if it is already compressed in the previous service), encodes it with ORC, and compresses using Zstd level 7. The data is destined for long-term storage, so a high compression ratio is favored over a high compression speed. The compute utilization of this service is dominated by Zstd compression, as it spends 28.5% of its compute cycles in compression and decompression as shown in Figure 6. Spark jobs involve reading data from Data Warehouse, running a set of workers, and then collating results and writing them back into Data Warehouse. **The Spark workers** (DW3) are reading

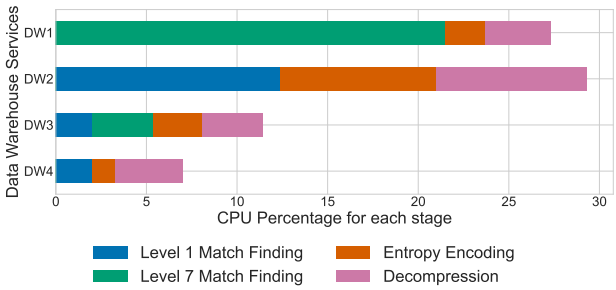


Fig. 7. Compression/decompression split of warehouse services.

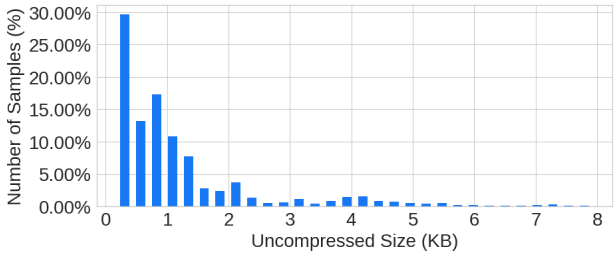


Fig. 8. Item size distribution for CACHE1.

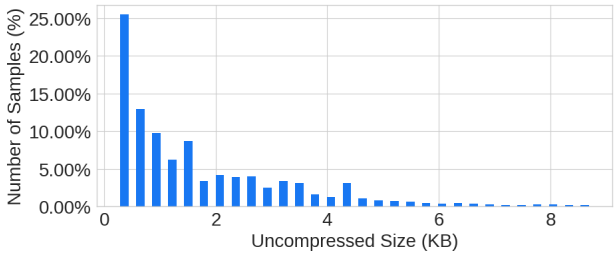


Fig. 9. Item size distribution for CACHE2.

the input data, decompressing it, doing their computation, and then re-writing to Data Warehouse. They spend 13.5% of the CPU in Zstd as shown in Figure 6. One specialized step of a Spark job is a **Shuffle** (DW2), which reads and decompresses the input data, then splits it by the destination worker, and writes the split data back into short-term storage with Zstd level 1 compression. It spends 22% of its compute cycles in Zstd level 1 compression, and 8% of its compute cycles in Zstd decompression, as shown in Figure 7. **Machine learning jobs** (DW4) consume data from Data Warehouse as model-related data. They spend 8% of their compute cycles in Zstd, split between level 1 compression and decompression. In Figure 7, we also split the compression time into match finding time and entropy encoding time. As explained Section II-B, the match finding stage is followed by the entropy encoding stage in Zstd. Usually, entropy encoding speed and decompression speed are independent of the compression levels, while a match finding speed heavily depends on the compression levels since a compression level changes parameters used for generating match candidates during the match finding stage. We observe that the match finding stage dominates the compute cycles (up to 80%) for DW1, where compression level 7 is mainly used, while match finding only takes around 30% of Zstd compute cycles of DW4.

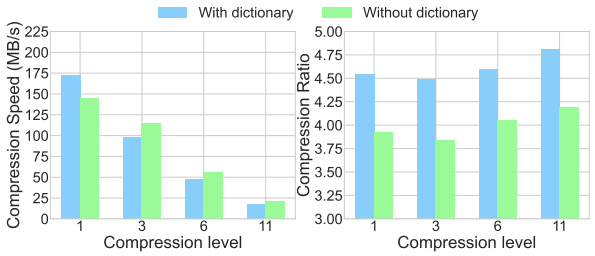


Fig. 10. Comparison with different compression level for CACHE1.

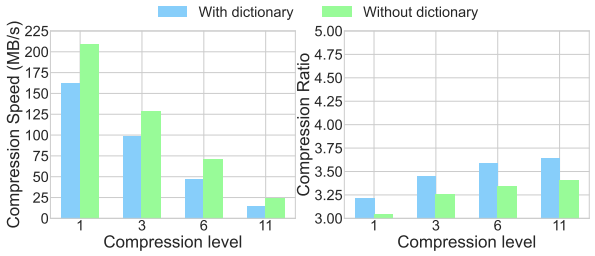


Fig. 11. Comparison with different compression level for CACHE2.

***Takeaway**: Compression takes a large portion of compute cycles at Data Warehouse, but the benefits of the reduced storage footprint could be worth the computational cost. The compression level needs to be tuned to balance compute and storage costs. It is worth spending more compute cycles to improve the compression of data destined for long-term storage.*

C. Caching services

Caching services are used to provide a shorter latency to access objects and reduce the loads to underlying databases. We focus on two important caching services CACHE1 and CACHE2 that are based on a popular distributed memory object caching system, Memcached [25]. Caches need to offer fast random access to their contents, so when they offer compression, they compress each item individually. This significantly reduces the compression ratio, but it can be re-gained by using **dictionary compression**. Compressing items individually means that the item can be sent compressed over the network to the client without decompressing on the server-side, saving both CPU and network. The client has to decompress the data, but the load is less centralized as each cache machine serves hundreds to thousands of clients. Data stored in CACHE1 and CACHE2 is typed, so we can group items by their type and provide one dictionary per data type, which drastically improves the compression ratio with dictionary compression.

We show the item size distributions for CACHE1 and CACHE2 in Figure 8 and Figure 9, respectively. We observe that the distribution is strongly skewed towards smaller items whose sizes are less than 1KB, with a long tail of larger items. Moreover, we measure the effectiveness of dictionary compression on the items stored in CACHE1 and CACHE2. In Figure 10 and Figure 11, we show the compression speed vs. compression ratio curves for CACHE1 and CACHE2 over compression levels 1, 3, 6, and 11 from right to left. We observe that dictionary compression achieves a much

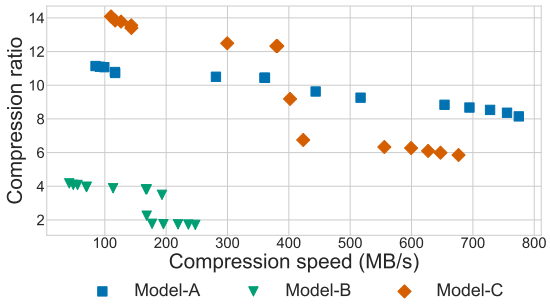


Fig. 12. Compression ratio and speed using various Zstd compression levels (from -5 to 9) with different models for ADS1.

higher ratio for the same level in all cases. Different levels of Zstd use different parameters/algorithms/heuristics such as the minimum acceptable length for a match. The trade-offs that a higher level makes tend to produce smaller compressed outputs with the lower speed, but there could be inconsistencies and some cases where these bets are wrong.

***Takeaway**: Dictionary compression can effectively improve compression ratio for small data. It is especially effective for cache item compression, where random access is important. Sending the data compressed over the network saves both cache CPU and network.*

D. Ads services

ADS1 service is a machine learning inference service that returns the target advertisement predictions through trained ranking models. An ADS1 service request is composed of a model input feature with metadata related to the target model. Since machine learning input features are usually large with frequent requests, transmitting them over the wire is expensive in terms of network footprint. Thus, compression of these service requests is a prospective optimization to reduce the associated network cost. For the trade-off, using compression requires compute overheads which incur extra latency (and some compute cost). Since this service has a strict latency requirement, it is important to understand the trade-off between the reduction in request size for reducing network traffic/cost and the increase in the application latency. The trade-off is also dependent on the data characteristics of a model request, which includes dense float and sparse integer embeddings. The ratio between different types of embeddings varies significantly between different models. Usually, higher compression ratios are achieved when compressing requests with more sparse embeddings due to the numerous zeros in the data. In Figure 12, we show how different model requests have different compression ratios and speeds depending on compression levels. Model A is the model which causes the most considerable traffic in the service with the largest average request size. Model B is another model generating high traffic with smaller size in requests. Model C is a different variant of model B using different serializations in the network. As shown in Figure 12, each model could use different compression configurations for optimization while meeting their requirements.

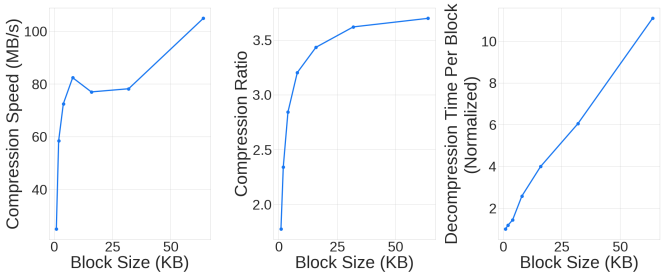


Fig. 13. Compression speed, compression ratio, and decompression time per block with different block sizes using sample data from KVSTORE1.

***Takeaway**: Compression can be used to reduce the requirement of network bandwidth for our ADS1 service, but the latency overhead should be considered carefully due to the strict latency requirement as well as model variance.*

E. Key-value store services

RocksDB [24] is a high-performance embedded persistent key-value store optimized for fast storage devices. It maintains an in-memory table to keep the key-value pairs, which gets stored in the file system as a Sorted Sequence Table (SST) file once it gets full. If the size reaches the capacity limit at each storage level, it runs compaction to reduce the size of SST files by compressing the files and removing overlapping items [20]. Usually, each SST file is broken into a number of blocks (with a block size of 16KB or 64KB) and compressed in a block granularity. KVSTORE1 is a high-performance distributed key-value store at Meta [36], which was developed based on RocksDB. A KVSTORE1 query is typically converted to one or more RocksDB queries. It also runs compaction which includes compression to reduce the storage footprint. The right configuration of compression (in terms of the specific algorithm, level, and block size) has direct bearings on the associated performance, i.e., compression ratio, space usage, and performance (compute, memory). An important requirement for KVSTORE1 is to meet a target read latency for the quality of the service. To read certain data in a block, the entire block needs to be decompressed since compression is done in a block granularity. Therefore, decompression time per block is important for the read latency of KVSTORE1. In Figure 13, we show how different block sizes affect compression ratio, speed, and decompression time per block. We used Zstd level 1 for this experiment using SST files with block sizes 1KB to 64KB. As we increase block sizes, we observe (usually) a higher compression ratio, speed, and decompression time. We also noticed that for smaller inputs, Zstd shrinks its hash tables, because there is little benefit to using a 1MB hash table to process 1KB of input. Shrinking the table will make the algorithm significantly faster because the working memory will sit in a faster cache. However, this is fighting against other compression costs that are fixed per-compression, which make very small compressions slower. So together, this leads to a non-monotonic performance profile. Internally, the KVSTORE1 team in Meta considers those trade-offs to find the sweet spot.

***Takeaway**: Using a larger block for compression could provide a higher compression ratio while incurring longer decompression time per block. Services that aim to reduce storage footprint while meeting a specific decompression time requirement like* KVSTORE1 *should consider these trade-offs.*

V. Sensitivity Studies

There are many options for compression that a user can choose such as different compression algorithms, compression levels, and block sizes. As Section IV showed, there cannot be a single "optimal" compression option that can be used by all services. Thus, engineers need to search for the best configuration for their service(s) considering different requirements along with different costs. Even, the best compression option for a particular service might change over time due to the changes in the data characteristics, application features, and compression libraries. For example, the storage cost of a service using Flash as its persistent store is different from that of a service using Hard Disk Drive as its persistent store. Also, some services have specific latency SLOs that can be impacted by compression and decompression speeds while others do not. Moreover, some services might have free CPU cycles to compress data while others need to postpone other threads to run compression. In this section, we introduce *CompOpt*, a compression optimizer based on analytical models to evaluate the costs of different compression choices. Subsequently, we show sensitivity studies using *CompOpt* to understand the impact of different compression configurations on the overall cost of production applications.

A. CompOpt design

CompOpt is a simple first-order optimizer that searches for the best compression option for a given service based on cost estimation and service requirements. *CompOpt*, as shown in Figure 14, takes sample data for the target service from a user as well as other service-specific costs and requirements. How to sample the data depends on the preference of the user; for example, if a user wants a more accurate estimation, then it would be better to sample large data. We introduce a module called CompEngine in *CompOpt* to generate different candidate compression options with different compression algorithms, compression levels, and block sizes. The current version of *CompOpt* supports several compressors including LZ4, Zlib, and Zstd. It can be easily extended to explore more compression parameters or support other compressors for different applications using the provided interfaces.

CompEngine runs candidate compression options with the sample data, which are then coupled with the corresponding compression ratio, compression speed, and decompression speed (referred as compression metrics). Using these compression metrics, the cost model (explained below) estimates the cost for each candidate compression option and *CompOpt* returns optimal compression configurations with estimated costs for the given service. With more compression parameters in the compression configuration, one might need to adopt efficient search methods based on random sampling, gradient-descent, or genetic algorithm, but the exhaustive search is sufficient for our study.

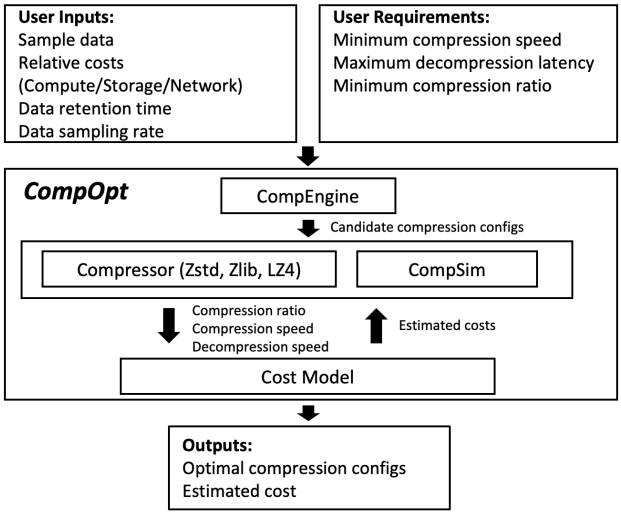


Fig. 14. The overview of *CompOpt*.

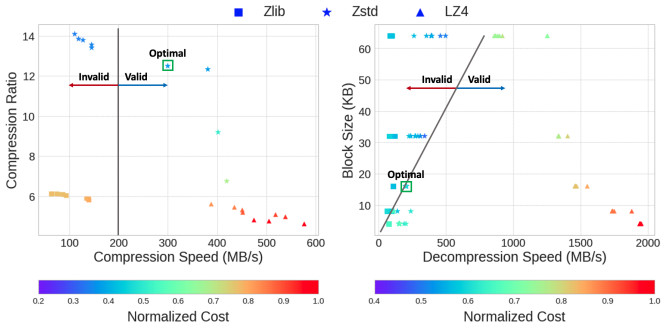
To estimate the overall cost which is the sum of compute, storage, and network cost, we first define a compression configuration x as a tuple composed of a compression algorithm, a compression level, and a block size, such as (Zstd, 3, 64KB) or (Zlib, 1, 16KB). Next, we use S to indicate a set of sample data used in the target service and s for each data in the set. We use the relative cost for compute, storage, and network as $\alpha_{compute}, \alpha_{storage}$, and $\alpha_{network}$ respectively, with the base cost B. We also denote the sampling rate, the ratio of the number of samples collected to the number of total function calls for compression in the target service, as β. We use R to indicate the average data retention time (days). The parameters introduced so far are assumed to be given by a user of *CompOpt*, and *CompOpt* uses them to estimate the costs of each compression configuration. Using the above parameters, the cost of compute, storage, and network ($c_{compute}(x)$, $c_{storage}(x)$, and $c_{network}(x)$) for a given compression configuration x are calculated as follows. The goal of *CompOpt* is to find the optimal compression configuration x_{opt}, which minimizes the overall cost, i.e. the sum of all costs.

$$c_{compute}(x) = \sum_{s \in S} \frac{\alpha_{compute} B \times Size(s)}{CompSpeed(x,s) \times \beta} \tag{1}$$

$$c_{storage}(x) = \sum_{s \in S} \frac{\alpha_{storage} B \times R \times Size(s)}{CompRatio(x,s) \times \beta} \tag{2}$$

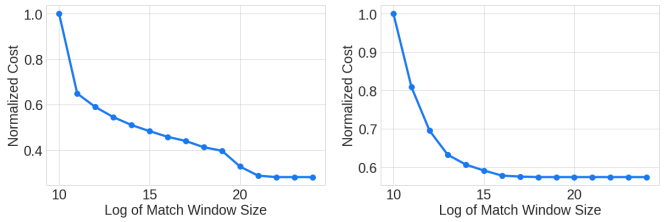
$$c_{network}(x) = \sum_{s \in S} \frac{\alpha_{network} B \times Size(s)}{CompRatio(x,s) \times \beta} \tag{3}$$

$$x_{opt} = \underset{x}{\operatorname{argmin}}(c_{compute}(x) + c_{storage}(x) + c_{network}(x)) \tag{4}$$



(a) Cost analysis for ADS1 (b) Cost analysis for KVSTORE1

Fig. 15. Normalized computational cost with different compression algorithms, levels, and block sizes for ADS1 and KVSTORE1.



(a) Cost analysis for ADS1 (b) Cost analysis for KVSTORE1

Fig. 16. Normalized costs with different match window sizes for ADS1 and KVSTORE1.

CompOpt also provides CompSim, an interface for future compression accelerator modeling, which enables exploring their benefits. Unlike the large flexibility in SW compressors, accelerator developers are often required to implement a HW-implementation-friendly variance of the compression algorithm in the hardware to minimize HW overheads. To understand the proper parameters for the target compression algorithm and the target service, HW developers can implement their simplified version of the compression algorithm in CompSim. Using CompSim, HW developers are able to understand the estimated compression ratio when used with their accelerators. To estimate the (de)compression speed of the target accelerator, the hardware designer can set a multiplication factor γ, which will be multiplied by the measured (de)compression speed. The HW designer can also set the $\alpha_{compute}$ for their accelerator to estimate the overall cost. *CompOpt* treats CompSim as another compressor when evaluating different compression configuration candidates.

B. Sensitivity studies using CompOpt

We conduct three sensitivity studies using *CompOpt* to show how different compression options affect to the overall costs that service owners and HW compression accelerator designers would like to minimize. We use Amazon EC2/EIA [1], [2] to estimate compute costs and Amazon S3 [3] to estimate storage and network costs.

Sensitivity study 1: Services like ADS1 target to minimize the cost of compute and network while storage cost is not important because the intermediate data is not stored. Since the service is latency-sensitive, the requirement is meeting a certain compression speed. Assuming that the minimum compression speed requirement as 200MB/s, we observed that **Zstd level-4** showed the lowest total cost, which is lower than 73% compared with the worst configuration (LZ4 with level 10) as shown in Figure 15 (a).

Sensitivity study 2: Services like KVSTORE1 target to minimize the cost of compute and storage while network cost is not important. It has flexibility for the block sizes but needs to meet a specific decompression latency for the read latency requirement. We assume the decompression latency requirement is given as 0.08ms. In Figure 15 (b), we show the estimated costs using different compression configurations with different block sizes (4/8/16/32/64KB). We observed that Zstd level-1 with 64KB showed the lowest total cost among all options, which is lower than 53% compared with the worst option (LZ4 level-1 with 4KB). If we consider the options meeting the given decompression latency requirement, **Zstd level-1 with 16KB** showed the lowest total cost, which is lower than 48% compared with the worst option.

Sensitivity study 3: In Figure 16, we explore the costs of different match window sizes using Zstd compression level 1 to find the optimal match window size to be implemented in the HW compression accelerator which is planned to be deployed for ADS1 and KVSTORE1 services by implementing the corresponding algorithms in CompSim. We use Amazon EIA cost as the compute cost of the accelerator and set γ as 10, but this can be set differently by the HW developers for their use cases. We observed that the normalized cost reaches the plateau around 2^{21}B and 2^{16}B for ADS1 and KVSTORE1, respectively, which indicates that different parameters for HW accelerators are preferable depending on the target workload.

VI. Future Research Directions

A. Compression SW

Compression algorithms (SW) is an area where many developers and researchers from Google [4], Meta [9], Microsoft [7], etc. are actively working on. From our characterization in Section IV, we have observed that many datacenter services are leveraging compression to minimize overall cost (storage, network, and compute) while satisfying different requirements (compression latency and decompression latency) for their target data that exhibit different characteristics. We have also found that the datacenter applications can be categorized into **A) Compression speed-sensitive** (which prefers low compression levels), **B) Decompression speed-sensitive** (which prefers small block sizes), **C) Latency-insensitive** (which prefers high compression levels), **D) Small data-friendly** (which prefers dictionary compression). We believe the future compression algorithms (SW) should be able to provide enough flexibility to handle all the categories to be suitable for diverse datacenter use cases. Furthermore, it is critical to develop up-to-date benchmarks which could represent datacenter workloads so that recent data characteristics such as block sizes could be carefully considered to properly estimate the impact of a new compression algorithm.

B. Compression HW

While a compression algorithm (SW) can provide many parameters to be tuned, it is impractical to implement all the features for the full flexibility that a compression algorithm provides in a specialized compression accelerator (HW) due to the area and power constraints. To mitigate the problem, the first direction is developing a highly-specialized and simple HW supporting a single sweet spot for a specific target workload using the prior knowledge or characterization about the workload such as the one provided in Section V. This option could minimize the HW cost, but would not be appropriate if the target workload does not show a specific pattern or the characteristics of the workload change over time. Developing a reconfigurable HW is another option that could adapt to changes while investing some HW cost for flexibility as trade-off. This approach would be appropriate if the characteristics of the workload change over time or if it has to be generic to cover various use cases (similar to compression SW). The fact that a compression algorithm (SW) itself also evolves over time reinforces the claim that developing a generic compression HW is more future-proof.

Regardless of HW design options, offloading overheads and bounding resources need to be considered carefully, which can often nullify the benefits of using compression HW. For example, services that belong to Category A and C mentioned above might prefer compression HWs to accelerate compression and reduce CPU burden while it would be better to run compression on CPU for Category B and D since offloading overhead would be significant for small blocks/data unless the accelerator is located very closely (such as on-chip).

C. Case for auto-tuners

Given the variety of choices among compression algorithms and hardware platforms, it is prohibitively expensive for a service engineer to manually conduct production experiments with each compression option to find the optimal point. Moreover, this is a task that should be done by all teams that use compression, so different teams could be spending time on duplicate work. Additionally, service characteristics often change over time. Hence, the optimal compression configuration is expected to change over time as it depends on data characteristics. We expect that there is a room for compression autotuners in this space that could act as a unified tool to find the optimal compression configuration for each service more efficiently and productively. As there are many hyperparameters for compression algorithms and different options for compression HW, the autotuner should be able to search over the huge space effectively while considering different service requirements similar to our *CompOpt* or Managed Compression [27]. The autotuners should be cost/SLO-aware instead of just focusing on naive compression metrics such as compression ratio or speed since the same ratio and speed could cause different overall costs.

VII. Related Work

Data compression has been widely used to reduce the cache, memory, and storage footprint [16], [30], [31], [42], [43]. A previous study [33] demonstrated about one quarter of overall tax cycles are spent compressing and decompressing data in Google data centers. The Accelerometer paper [40] showed that 3-15% of compute cycles are spent (de)compression in Facebook datacenters. However, they do not provide detailed fleet-level and service-level characterization of compression, which is one of the main contributions of our work.

Prior works related to compression characterization have mostly focused on evaluating the efficacy of various compression schemes with benchmark data sets [17], [19], [23], [38]. To the best of our knowledge, none of the previous works has explored the workloads of datacenters from the perspective of compression. We present the first work to characterize the finer aspects of compression with real application data in representative warehouse-scale datacenters.

Compression algorithms are SW techniques for compression, which can be broadly classified into lossless and lossy. Lossless compression provides a one-to-one mapping between original data and compressed data enabling accurate regeneration of compressed data [38], thus being widely adopted in datacenter services. There are various algorithms available for lossless data compression such as Zstd, LZ4, and Zlib. Hardware acceleration for data compression [10]–[12], [15], [26], [32] is another promising technique that can provide an order of magnitude improvement in compression speeds and help free up compute cycles. In this work, we provide a tool, *CompOpt*, which can estimate the overall cost using a compression hardware or compression algorithm with various compression configurations to understand their impact and trade-offs.

VIII. Conclusion

This work provides a comprehensive characterization of compression (both at the datacenter-level and service-level) at a global social media provider. We found compression amounts to an overall 4.6% compute cycles while some services spend as much as 30% of their compute cycles in compression. Along with the detailed compression characterization study of datacenter applications, we provide insights into various trade-offs for compression. We also develop a tool, *CompOpt*, to estimate performance-cost trade-offs with different compression options, followed by sensitivity studies using this tool to understand the impact of compression configurations. We believe our work paves a way for developing efficient compression HW and SW for datacenter services.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback to improve our paper. We also appreciate the engineers and researchers at Meta for their insightful comments and suggestions.

REFERENCES

[1] "Amazon ec2 on-demand pricing," accessed: 2023-03-21. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/

[2] "Amazon elastic inference pricing," accessed: 2023-03-21. [Online]. Available: https://aws.amazon.com/machine-learning/elastic-inference/pricing/

[3] "Amazon s3 pricing," accessed: 2023-03-21. [Online]. Available: https://aws.amazon.com/s3/pricing/?nc=sn&loc=4

[4] "Brotli website," accessed: 2023-03-21. [Online]. Available: https://github.com/google/brotli

[5] "Fse website," accessed: 2023-03-21. [Online]. Available: https://github.com/Cyan4973/FiniteStateEntropy

[6] "Lz4 website," accessed: 2023-03-21. [Online]. Available: https://lz4.github.io/lz4/

[7] "Zipline website," accessed: 2023-03-21. [Online]. Available: https://github.com/opencomputeproject/Project-Zipline

[8] "Zlib website," accessed: 2023-03-21. [Online]. Available: https://www.zlib.net/

[9] "Zstd website," accessed: 2023-03-21. [Online]. Available: https://facebook.github.io/zstd/

[10] "Improved cloud service performance through asic acceleration," 2019, https://azure.microsoft.com/en-us/blog/improved-cloud-service-performance-through-asic-acceleration/.

[11] "Intel quickassist technology (qat) qatzip library," 2019, accessed: 2023-03-21. [Online]. Available: https://github.com/intel/QATzip

[12] B. Abali, B. Blaner, J. Reilly, M. Klein, A. Mishra, C. B. Agricola, B. Sendir, A. Buyuktosunoglu, C. Jacobi, W. J. Starke, H. Myneni, and C. Wang, "Data compression accelerator on ibm power9 and z15 processors : Industrial product," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 1–14.

[13] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "Gzip on a chip: High performance lossless data compression on fpgas using opencl," in *Proceedings of the International Workshop on OpenCL 2013 & 2014*, ser. IWOCL '14. Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2664666.2664670

[14] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed, "Scuba: Diving into data at facebook," *Proceedings of the VLDB Endow.*, p. 1057–1067, 2013. [Online]. Available: https://doi.org/10.14778/2536222.2536231

[15] K. Aingaran, S. Jairath, and D. Lutz, "Software in silicon in the oracle sparc m7 processor," in *2016 IEEE Hot Chips 28 Symposium (HCS)*, 2016, pp. 1–31.

[16] A. Alameldeen and D. Wood, "Adaptive cache compression for high-performance processors," in *2004 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004, pp. 212–223.

[17] R. Arnold and T. Bell, "A corpus for the evaluation of lossless compression algorithms," in *Proceedings DCC '97. Data Compression Conference*, 1997, pp. 201–210.

[18] L. A. Barroso, U. Hölzle, P. Ranganathan, and M. Martonosi, *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Morgan & Claypool, 2018.

[19] T. Bell, I. H. Witten, and J. G. Cleary, "Modeling for text compression," *ACM Comput. Surv.*, p. 557–591, 1989. [Online]. Available: https://doi.org/10.1145/76894.76896

[20] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in *2020 18th USENIX Conference on File and Storage Technologies*, ser. FAST'20. USENIX Association, 2020, p. 209–224.

[21] J. Chen, M. Daverveldt, and Z. Al-Ars, "Fpga acceleration of zstd compression algorithm," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2021, pp. 188–191.

[22] Y. Deguchi, A. Kobayashi, H. Watanabe, and K. Takeuchi, "Flash reliability boost huffman coding (frbh): Co-optimization of data compression and vth distribution modulation to enhance data-retention time by over 2900x," in *2017 Symposium on VLSI Technology*, 2017, pp. T206–T207.

[23] S. Deorowicz, "Universal lossless data compression algorithms," Ph.D. dissertation, Silesian University of Technology, 2003.

[24] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in rocksdb," in *2017 8th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2017. [Online]. Available: http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf

[25] B. Fitzpatrick, "Distributed caching with memcached," *Linux Journal*, 2004.

[26] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, "A scalable high-bandwidth architecture for lossless compression on fpgas," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015, pp. 52–59.

[27] F. Handte, Y. Collet, and N. Terrell, "Zstandard," accessed: 2023-03-21. [Online]. Available: https://engineering.fb.com/2018/12/19/core-data/zstandard/

[28] D. Harnik, E. Khaitzin, D. Sotnikov, and S. Taharlev, "A fast implementation of deflate," in *2014 Data Compression Conference*, 2014, pp. 223–232.

[29] X. He, J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, S. Bowers, and J. Q. n. Candela, "Practical lessons from predicting clicks on ads at facebook," in *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, ser. ADKDD'14. Association for Computing Machinery, 2014, p. 1–9. [Online]. Available: https://doi.org/10.1145/2648584.2648589

[30] S. Hong, P. J. Nair, B. Abali, A. Buyuktosunoglu, K.-H. Kim, and M. Healy, "Attaché: Towards ideal memory compression by mitigating metadata bandwidth overheads," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 326–338.

[31] X. Hu, F. Wang, W. Li, J. Li, and H. Guan, "Qzfs: Qat accelerated compression in file system for application agnostic and cost efficient data storage," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USENIX Association, 2019, p. 163–176.

[32] Intel, "Intel quickassist technology api programmer's guide," 2021, accessed: 2023-03-21. [Online]. Available: https://01.org/sites/default/files/downloads//330684-011-intel-qat-api-programmers-guide.pdf

[33] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," *SIGARCH Comput. Archit. News*, p. 158–169, 2015. [Online]. Available: https://doi.org/10.1145/2872887.2750392

[34] J. Y. Kim, S. Hauck, and D. Burger, "A scalable multi-engine xpress9 compressor with asynchronous data transfer," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014, pp. 161–164.

[35] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan, "Software-defined far memory in warehouse-scale computers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2019, p. 317–330. [Online]. Available: https://doi.org/10.1145/3297858.3304053

[36] S. Masti, "How we built a general purpose key value store for Facebook with ZippyDB," accessed: 2023-03-21. [Online]. Available: https://engineering.fb.com/2021/08/06/core-data/zippydb/

[37] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, and K. Sheng, "Fpga implementation of gzip compression and decompression for idc services," in *2010 International Conference on Field-Programmable Technology*, 2010, pp. 265–268.

[38] L. Promberger, R. Schwemmer, and H. Fröning, "Characterization of data compression across cpu platforms and accelerators," *Concurrency and Computation: Practice and Experience*, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6465

[39] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner, "Presto: Sql on everything," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019, pp. 1802–1813.

[40] A. Sriraman and A. Dhanotia, "Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2020, p. 733–750. [Online]. Available: https://doi.org/10.1145/3373376.3378450

[41] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*, 2015, pp. 583–590.

[42] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, and D. Skarlatos, "Tmo: Transparent memory offloading in datacenters," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2022, p. 609–621. [Online]. Available: https://doi.org/10.1145/3503222.3507731

[43] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *2000 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2000, pp. 258–265.

[44] Y. Yang, J. S. Emer, and D. Sanchez, "Spzip: Architectural support for effective data compression in irregular applications," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1069–1082.

[45] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, pp. 337–343, 1977.