DISSERTATION

submitted to the

Combined Faculty of Mathematics, Engineering and Natural Sciences

of the

Ruprecht–Karls University
Heidelberg

for the degree of

Doctor of Natural Sciences

put forward by

Laura Promberger, M.Sc.

born in
Dachau, Bavaria, Germany

May, 2022

Compute, Storage and Throughput Trade-offs for High-Energy Physics Data Acquisition

Advisor: Professor Dr. Holger Fröning

Date of oral exam: ....

And on the map was written
Here Be Dragons

Acknowledgements

This work would not have been possible without all the support I received from various marvelous people.

I, therefore, would like to thank Prof. Dr. Holger Fröning for the willingness to supervise this project in collaboration with CERN and for the trust he put in me to be successful. His guidance significantly influenced my professional and personal growth, making me a better researcher and reducing the impact of prolonged setbacks.

Special thanks to Rainer Schwemmer and Niko Neufeld for being my supervisors at CERN. Their expertise in many areas of computer science and physics is phenomenal and gave me opportunities to outgrow myself in many more areas than it would have been possible in other environments.

Naturally, I want to thank my wonderful parents that fed me through all those years and helped me grow into who I am. Their belief in me, the opportunities they offered me, and their support to go my (unusual) path gave me strength in many moments. Thank you for sparking my interest in science with all the weekly visits to the *Deutsche Museum* in Munich and watching the TV-Show *Die Sendung mit der Maus* when I was little.

I would also like to thank the many people of EP-LBC with whom I had many fruitful discussions and who I am glad to call my friends. Notable mentions are needed for Daniel, Tommaso, Francesco, and Flavio. Furthermore, thanks to my fellow Ph.D. students Bernhard, Kevin, Lorenz, and Günther in Holger's group with whom I had many thoughtful and productive conversations. Also, many thanks to my *normal* friends that endured and supported me during this journey: Lukasz, Susan, Kerstin, Stephan, and Max.

Last but not least, I would like to acknowledge the funding I received, without which this work would not have been possible: *This work has been sponsored by the Wolfgang Gentner Programme of the German Federal Ministry of Education and Research (grant no. 13E18CHA).*

Abstract

Nowadays, the large majority of research insights are gained by using compute-aided analyses. Before the analysis, data needs to be acquired and prepared. Depending on the source of data, its acquisition can be a complex process. In that case, so-called Data Acquisition Systems (DAQs) are used. Real-time DAQs require unique challenges to be solved, either in latency or throughput. At the European Organization for Nuclear Research (CERN), where High Energy Physics (HEP) experiments collide particles, real-time DAQs are deployed to filter down the vast amount of data (for LHCb experiment: up to 4 TB/s). When working with large amounts of data, data compression allows improving limitations in capacity and transfer rates. This work is about the various compression techniques that exist and their evaluation for real-time HEP DAQs.

The first part characterizes popular general-purpose compression algorithms and their performance using ARM aarch64, IBM ppc64le, and Intel x86_64 CPU architectures. Their performance is found to be independent of the underlying CPU architecture, making each architecture a viable choice. The scaling and robustness are dependent on the number of simultaneous multithreading (SMT) available. High numbers of SMT scale better but are less robust in performance. When it comes to "green" computing, ARM outperforms IBM by a factor of 2.8 and Intel by a factor of 1.3.

The second part designs a co-scheduling policy that improves the integration of compression devices. This policy allows for efficient and fair distribution of performance between (independent) host and device workloads. It only needs two metrics: power consumption and memory bandwidth, and does not require any code changes for the host workload. Solely with NUMA binding and either polling or interrupts for communication with the device, the performance increases for resource-unsaturated host workloads by a factor of 1.5 – 4.0 for the device and by a factor of 1.8 – 2.3 for the host. For resource-saturated host workloads, it increases by a factor of 1.8 – 1.9 for the device but decreases by 0.1 – 0.4 for the host.

The third part evaluates two compression techniques utilizing domain-based knowledge: Huffman coding and lossy autoencoders. Huffman coding on the original data compresses 40 – 260% better than any tested general-purpose algorithms. Huffman coding on delta encoded data performs poorly for HEP data. Autoencoders are a popular machine learning

technique. Two data representations, including One Hot Encoding, and many hyperparameters are tested. However, all configurations turn out to compress too lossy. They need more technological advances to improve the performance of neural networks with large layers.

And the last part performs a cost-benefit analysis of the previously presented compression techniques. It is based on power savings and capital expenses. Applied to the real-time LHCb DAQ, it concludes that only compression accelerators are an economically viable choice. Huffman coding on absolute values achieves a higher compression ratio than any general-purpose solution but is too slow. More research would be needed to find a better fitting compression technique based on domain knowledge.

While the context of this work is real-time DAQs in the HEP community with specific requirements and limitations, we believe the results of this work are generic enough to apply to the majority of environments and data characteristics.

Zusammenfassung

Heutzutage wird der Großteil von Forschungserkenntnisse mit Hilfe von computerunterstützenden Analysen gewonnen. Dafür müssen Daten erworben und für die Analyse vorbereitet werden. Abhängig von der Datenquelle kann dies so komplex sein, dass so genannte Data Acquisition Systems (DAQs) benutzt werden müssen. Echtzeit-DAQs zeichnen sich durch besondere Anforderungen an Latenzzeiten oder Datendurchsatz aus. An der Europäische Organisation für Kernforschung (CERN) werden Experimente in der Hochenergiephysik (HEP) durchgeführt, bei denen Teilchen mit nahezu Lichtgeschwindigkeit miteinander kollidiert werden. Echtzeit-DAQs werden dort eingesetzt um die großen Datenmengen (für LHCb Experiment: 4 TB/s) bereits bei der Datenextraktion zu filtern und zu reduzieren. Datenkompression ermöglicht es bei so großen Datenmengen die Limitierung bezüglich Speicherplatz und Durchsatzraten zu verbessern. Diese Arbeit hier handelt von verschiedenen Datenkompressionsmethoden und deren Nutzbarkeit in HEP Echtzeit-DAQs.

Im ersten Teil werden verlustfrei Universalkompressionsalgorithmen und ihre Leistungsfähigkeit auf unterschiedlichen CPU-Architekturen (ARM aarch64, IBM ppc64le, Intel x86_64) bewertet. Es zeigt sich, dass ihre Leistung unabhängig von der unterliegenden CPU-Architektur ist. Somit ist jede CPU-Architektur eine gute Wahl. Ihre Skalierbarkeit und Leistungsrobustheit ist abhängig davon, wie viele Threads pro CPU-Kern existieren. Eine hohe Threadanzahl skaliert besser, ist aber auch weniger robust. Bezüglich "Green" Computing ist ARM 2.8-mal besser als IBM und 1.3-mal besser als Intel.

Im zweiten Teil wird eine Co-scheduling Methode entwickelt, die eine effizientere Nutzung von Kompressionsbeschleunigern ermöglicht. Die Methode erlaubt es effizient und fair die Leistungsfähigkeit zwischen (unabhängiger) Host- und Beschleunigersoftware zu verteilen. Zwei Parameter werden benötigt: Stromverbrauch und Speicherbandbreite. Besonders vorteilhaft ist, dass der Quellcode von der Hostsoftware nicht geändert werden muss. Alleine mit NUMA-Binding und dem richtigen Kommunikationsschema für den Beschleuniger, ist es möglich, die Leistung von einer nicht-ressourcen-limitierten Hostsoftware um den Faktor 1.8 – 2.3 zu erhöhen und gleichzeitig auch die Leistung für den Beschleuniger um 1.5 – 4.0 zu erhöhen. Für ressourcen-limitierte Hostsoftware verringert sich die Leistung um einen Faktor 0.1 – 0.4, während der Beschleuniger eine Leistungsverbesserung von 1.8 – 1.9 erfährt.

Im dritten Teil werden zwei Kompressionsmethoden evaluiert die Domänenwissen benutzen: verlustfreies Huffman-Kodierung und verlustbehaftete Autoencoders. Huffman-Kodierung für Absolutwerte komprimiert 40 – 260% besser als jeder Universalkompressor. Huffman-Kodierung für Deltawerte funktioniert nicht gut für HEP Daten. Autoencoders sind eine populäre Machine-Learning (ML) Methode. Zwei Datendarstellungen, darunter One Hot Encoding, und viele Hyperparameter werden getestet. Jedoch komprimieren alle Kombinationen mit zu großen Verlusten. Mehr technologischer Fortschritt ist notwendig, um bessere Ergebnisse für große ML-Netzwerke zu erreichen.

Im letzten Teil wird eine Kosten-Nutzen-Analyse von den zuvor vorgestellten Kompressionsmethoden durchgeführt. Sie analysiert Energieersparnis und Anschaffungskosten. Für das Echtzeit-DAQ vom LHCb Experiment ist die einzige profitable Lösung die Nutzung von Kompressionsbeschleunigern. Huffman-Kodierung für Absolutwerte erreicht eine bessere Kompression als die Universalkompressoren, jedoch ist sie zu langsam für das DAQ. Mehr Forschung wäre nötig, um eine besser passende Kompressionsmethode zu finden, die Domänenwissen benutzt.

Der Kontext dieser Arbeit sind Echtzeit-DAQs in HEP, die besondere Anforderungen und Limitationen haben. Gleichwohl sind wir davon überzeugt, dass die Ergebnisse dieser Arbeit universal anwendbar und in vielen verschiedenen Umgebungen und für verschiedene Datencharakteristiken gültig sind.

Table of contents

Chapter 1

Introduction

The digitalization in the current age has advanced so much that the large majority of research insights are gained by using compute-aided analyses. Any compute-based analysis that uses external data, like sensor data or data scraped from the internet, needs processes to acquire, prepare, and finally analyze that data. This order of steps can be described as a pipeline. For example, in data science, the OSEMN pipeline [63] consists of the following steps: 1) Obtain the data, 2) Scrub/Clean the data, 3) Explore the data, 4) Model prediction based on the data and 5) Interpretation of the results. Depending on the source of data, it can be a complex endeavor to acquire it. In such cases, entire systems, so-called Data Acquisition Systems (DAQs), are needed. The usage of DAQs often implies Big Data sources that continuously create vast amounts of data that need to be automatically processed. Unique challenges are created if the DAQ needs to function in real-time, either because immediate feedback is required (e.g., self-driving cars) or because the data source creates too much data. In the second case, challenges include bandwidth limitations and storage limitations. They make it necessary that data cleaning and filtering procedures must already be part of the DAQ. This is the case at the European Organization for Nuclear Research (CERN), where High Energy Physics (HEP) experiments collide particles. The high frequency at which the collisions are created results in vast amounts of data being created. For instance, in the LHCb experiment, data is acquired at 4 TB/s, but only about 0.3% of that data is stored permanently for analyses.

When handling and storing data, data compression becomes a natural next step in managing the vast amounts of data. Data compression allows reducing the size to represent data either 1) without losing any information (lossless compression) or 2) with limited information loss (lossy compression). With the ongoing technological advancements that make compute operations cheaper than data transfer operations, it becomes increasingly attractive to deploy compression and invest in the trade-off analysis of compute power, compute time, and storage capacity. As such, compression is nowadays not only limited to its traditional fields, including image visualization and database management, but it is also becoming a fundamental part of any kind of field working with Big Data. Depending on the field, preference exists for either lossless or lossy compression. For example, lossy compression is an accepted standard for image compression, like it is used in the JPEG2 algorithm. It can achieve significantly better compression than lossless compression, and the human eye can compensate for quite a lot of information loss without falsifying the information within the image. HEP research, on the other hand, traditionally only accepts lossless compression as already capturing the physical data by sensors introduces losses. This makes any additional loss of information undesirable. The newfound interest in compression can also be seen in the Machine Learning (ML) community, where it is an art to compress large neural networks without efficiency deterioration [105]. And it is also seen by NVIDIA that provides with nvcomp [82] a dedicated compression library for GPUs, even though architecture-wise, they are not ideal for executing compression [36]. The usage of data compression in real-time DAQs and the respective system design choices and trade-offs between compute, storage, and throughput becomes therefore an interesting application.

The majority of research in data compression is focused on developing new algorithms or implementing popular algorithms on accelerator hardware. This ranges from algorithms created by only a few people like lz4 [72] or bzip2 [55] or algorithms created by big cooperations like Facebook's zstd [115] or Google's snappy [43]. Accelerator support includes SIMD on CPUs [67], GPUs [10], FPGAs [38] and ASICs. In general, in such works, the aspect of integrating data compression into a system is neglected. While works exist for co-scheduling device and host workloads [12, 8, 99], the combination with data compression is rare. In addition, such works tend to present tightly-coupled frameworks which expect an overarching goal of device and host workloads [26].

This work aims to provide answers to reduce this existing gap in understanding the cost of system integration for data compression - the trade-offs between compute, storage, and throughput and the implications on the system design. In particular, we will analyze the trade-offs when dealing with data from physics detectors. The following contributions will be presented

1. A thorough characterization across CPU platforms of frequently used general-purpose lossless compression algorithms (`bzip2`, `lz4`, `snappy`, `zlib`, `zstd`, `xz`). One server for each CPU platform: ARM aarch64, IBM ppc64le, and Intel x86_64 of different price classes and release years. The characterization includes the metrics: cross-platform behavior, performance per core (unified metric across the platforms), robustness, scalability, and power consumption and efficiency. (Chapter 4)

2. A policy that provides an efficient and fair co-scheduling between independent host and (compression) device workloads. This policy requires no in-code changes to the host workload. It works by using NUMA-binding and selecting either polling or interrupts as communication schema for the device workload. The selection is based on two easily available performance metrics: memory bandwidth and CPU power consumption. (Chapter 5)

3. An exploration of multiple compression algorithms applied and trained on LHCb detector data to benefit from domain knowledge. Algorithms include a lossless Huffman coding on two data representations and different versions of lossy autoencoders, a machine learning technique. (Chapter 6)

4. A cost-benefit analysis of each of the compression options presented to evaluate their viability for the integration into the LHCb DAQ. (Chapter 7)

Overall, this work will provide approaches for integrating data compression into a system. Through multiple chapters, we will shed light on the trade-offs between compute, storage, and throughput and the implications on the system design when deploying data compression; the advantages and disadvantages of using general-purpose compression algorithms; how to efficiently and fairly integrate accelerators; and if this is all not viable, how to adapt and optimize existing compression algorithms to your custom data. This is all done in the context of real-time DAQs in the HEP community that have specific throughput and storage requirements and limitations. But, we believe the results of this work are generic and applicable to the majority of environments and data characteristics.

Preview

This work consists of 10 chapters. The current introductory chapter is followed by two auxiliary chapters, chapter 2: Related Work, which presents the recent state-of-the-art research, and chapter 3: Background, which offers background information about CERN, the LHCb experiment, a short mathematical introduction to information theory, and data compression algorithms and software libraries. They are followed by four core chapters, chapters 4 - 7, each presenting one contribution. The chapters 8: Discussion, 9: Conclusion and 10: My Publications complete and summarize this work.

Contribution 1: Cross-Platform Performance

The first contribution, the characterization of data compression libraries across CPU platforms, is presented in chapter 4. We will learn that the choice of CPU platform neither influences the compression ratio nor the relative throughput performance between algorithms A and B. However, the choice of data set influences the relative throughput performance and the compression ratio. Evaluating further parameters, we learn that performance robustness and scalability are related to the threads per core. More threads per core mean better scalability but also less robustness. For power efficiency or "green" computing, ARM outperforms Intel and outperforms even stronger IBM. A metric is introduced that allows the comparison of servers with different architectures and core counts.

This contribution concludes that all three platforms are equally viable for general-purpose compression algorithms. Therefore, the platform selection should be based on the cost-benefit analysis and the requirements for the rest of the system.

Contribution 2: Co-scheduling Policy

The second contribution, the co-scheduling policy, is presented in chapter 5. The policy we develop is based on the per-socket memory bandwidth and CPU power consumption of the standalone performance of the host and device workloads. There are four possible outcomes, with the principal decision taken based on the memory bandwidth needs of host and device workloads. Case 1: If the memory bandwidth needs are high, the NUMA node to which the device is connected should be avoided for the host workload. Polling is the preferred communication schema unless the host workload has a high power consumption. Then, interrupts should be used. Case 2: If the memory bandwidth needs are significantly higher than the socket provides, the entire socket should be avoided for the host workload, and no preference exists for the communication schema. Case 3: The memory bandwidth needs are low enough, but the power consumption of the host workload is high. Then interrupts should be used for the device communication. Case 4: Both the memory bandwidth needs and the power consumption are low enough. The choice of communication schema then depends on if the host workload has priority. If yes, use interrupts; if no, use polling.

In simple cases, when either the memory bandwidth needs or the power consumption is the limiting factor, the policy gains significant performance for both device and host workload. In complex cases, when memory bandwidth and power consumption are both limiting factors, the policy gains significant performance for the device, but the host workload loses performance. The loss of the host performance is less than half of the performance gain for the device.

Contribution 3: Compression With Domain-Knowledge

The third contribution, exploring and evaluating compression that utilizes domain knowledge, is presented in chapter 6. It consists of two parts which are both evaluated on the subdetector data VELO of the LHCb raw detector data. VELO data can be divided into three sections: *Header*, *Cluster Centroids*, and *ADC values*.

Part one is about the lossless Huffman coding on the data representations *delta encoded* and *absolute values*. Delta encoded means that between two VELO events, the delta of each field is encoded. Huffman coding on *absolute values* outperforms *delta encoded* data and all tested general-purpose compression algorithms by 40 – 260% in compression performance. In its throughput, it is 13% faster than the *delta encoded* data or right in the middle of the throughput of `bzip2 level 9` and `xz level 1`.

Part two is about autoencoders on the data representations *padded* (SLE) and *one-hot encoded* (OHE). Many hyperparameters for optimizer, loss function and activation function are tested to evaluate the best performance. On average, SLE achieves the best performance with Adam, MSE, ReLU, and OHE with Nadam, Huber, or Log Cosh, ReLU. Both autoencoders result in lossy compression. SLE has its best reconstruction results for *Header* and *Cluster Centroids* and worst for *ADC values*. OHE has perfect reconstruction for *Header*, small error for *ADC values*, and cannot reconstruct *Cluster Centroids*. In addition, it suffers from memory limitations resulting in only being trained on 1% of the data.

Contribution 4: Cost-Benefit Analysis of Compression for the LHCb DAQ

The fourth contribution, the cost-benefit analysis of different compression techniques for the real-time LHCb DAQ, is presented in chapter 7. It covers the analysis based on power savings and capital expenses. For the general-purposes compression algorithms only `snappy` and `lz4 level 2` would satisfy the hardware requirements. But power savings and capital expenses result in a net-negative evaluation. For compression accelerators, three different solutions would result in a positive outcome. Huffman coding on absolute values achieves the best compression ratio but has the same problem as most general-purpose compression algorithms: it is too slow. Autoencoders are not considered as they compress lossy.

As such, the only solution that fulfills all requirements and constraints of the LHCb DAQ are compression accelerators.

Chapter 2

Related Work

This chapter will give us an overview of current progress and research interests in the field of data compression. It is most relevant for chapters 4 and 5.

Data compression is traditionally used in two domains due to the amount of data: image and video processing [42], and databases [116]. With the advancement of Big Data in other domains, compression also becomes of interest there. This ranges from using compression in mobile devices [78], increasing the lifetime in SSDs [66] or communicating between robots [74]. Compression is also relevant to the scientific community. For example, Zeyen et al. [113] analyze lossy and lossless compression for data of cosmic particles. And also at CERN experiments are already using compression, e.g. ALICE [97] uses Huffman coding and LHCb [47] zero-suppression[1].

Research in data compression has two strong foci: 1) Developing new compression algorithms and 2) Optimizing existing algorithms for accelerator hardware like FPGAs, ASICs, or GPUs.

The first case often includes performance comparisons to existing and well-established algorithms to promote the new algorithm. This is, for example, the case for the better-known algorithms brotli [6] and lz4 [72] where their algorithm is compared against at least the algorithms zlib, zstd and snappy.

And in the second case, the focus lies on the performance difference, both compression speed and compression ratio, for the different implementations of the same algorithm. The approaches for accelerating compression include utilizing vectorization on the CPU using SIMD intrinsics [67], and using GPU [10], FPGA [38] or ASIC devices as compression accelerators. The majority of those works focuses on accelerating compression algorithms of the LZ77 and LZ78 family. The defacto industry standard *deflate* and its file formats zlib and gzip are the all-time favorites to implement. For example, Abdelfattah et al. [3]

[1]Compression by suppression of coordinates that did not register particles

implement gzip on an FPGA using OpenCL, Plugariu et al. [86] implement gzip on an FPGA integrated into a Hadoop cluster, Ozsoy et al. [83] implement lzss for GPUs using CUDA, and NVIDIA's compression library nvcomp [82] offers five algorithms of which one is an LZ77 variant called GDeflate. Furthermore, Fowers et al. [38] provide a modified version of deflate.

The choice of accelerating device also has a clear favorite: FPGA. FPGAs are reprogrammable chips that allow the highly optimized implementation of algorithms with high throughput and low-energy costs directly in hardware. Their reprogrammability and performance efficiency makes them attractive to researchers and an ideal prototyping platform before switching to more cost-efficient but hard-wired ASICs. These benefits prevail over the disadvantages of being cost- and time-intensive and developers needing to have sound knowledge of high-level requirements of the algorithm and low-level limitations of the hardware. The limitations include the number of logic gates available, the amount of memory available, and the, in comparison, very slow bandwidth connection between host and device. In addition, there is the complex routing required to connect the logic gates on the chip. Nevertheless, the largest majority of research work on accelerating compression uses FPGAs. One of the reasons is their excellent behavior for integer computations. Something also highly used in compression. However, the disadvantages of compression on FPGAs lie in the inherent nature of compression. Compression is pattern recognition. For this, we need two things: historical data and an intelligent decision, or in other words: memory and logic gates. Both are limited on an FPGA [92]. Thus, developing well-performing compression algorithms on FPGAs has proven to be challenging. The works here include the usage of FPGAs to accelerate database analytics while also integrating compression on the FPGA [106], using FPGA compression to increase the lifetime of SSDs [66] or implementing in OpenCL fractal video compression [20]. The significance of deflate is also shown by the commercial products of IBM's on-chip deflate algorithm for IBM Power9 and z13 (IBM's mainframe) [2], Intel's QuickAssist solutions [53] using ASICs and the data compression accelerators by the AHA Product Group [46].

While those two foci are very research-centered, they often lack the step of analyzing how their solutions perform outside the sandbox. For example, only a few works are about a dedicated performance comparison of existing compression algorithms in different environments. One of those studies, Matai et al. [75], analyzed the energy efficiency of canonical Huffman coding for Intel, ARM, and FPGA, with Intel being the least efficient and the FPGA the most efficient energy-wise. However, most of those few studies tend to be limited in the number of algorithms selected and in the number of data sets selected. The most common algorithms included are deflate and bzip2. The data sets included are often traditional compression corpora, like Calgary corpus [4] or Silesia corpus [4]. Those traditional compression corpora were established in the 90s and early 2000s and are therefore small in data size and mainly consist of text data. This type of data is not a diverse representation of Big Data. Furthermore, if different hardware performance is compared in most cases, it is a comparison of an (Intel) CPU platform with FPGA, ASIC, or GPU. This is the case for the works by Chen et al. [20] that compare fractal video compression implemented in OpenCL on three different platforms: an Intel Xeon, an NVIDIA GPU, and a Stratix IV FPGA. The same applies to Abdelfattah et al. [3] gzip OpenCL implementation, which compares its performance on the Calgary corpus to the performance of another FPGA, ASIC, and Intel CPU. Or, the works by Krishna et al. [60] that analyzes the hardware acceleration capabilities of the IBM PowerEN processor for zlib in comparison to the software implementation on the Canterbury corpus.

The results of such performance comparison of algorithms can provide input for practical applications, like the relatively recent work from 2019 by Devarajan et al. [31] that introduces an intelligent, adaptive, and flexible data compression framework. They propose a framework that dynamically selects the best compression library adapting to the data input and the given requirements where it is deployed. Contrary to our work in chapter 4, the framework explores its performance efficiency only on one server, and their choice of generated scientific data has a higher compression ratio than the HEP data we use. Their scientific data has a compression ratio similar to text data.

Coming back to FPGAs, another topic lacking in compression research is the system integration and its optimization of performance when co-scheduling host and devices. In general, simply installing accelerators in a system and expecting it to maximize the overall system performance will not work. Instead, an efficient co-scheduling between host and device workloads is needed. Different approaches provide answers to this question. However, they tend to be high-level approaches that require in-code changes for both, host and device workloads, and as such, making it unattractive to deploy in existing systems. For example, Belwal et al. [12] analyze different hybrid solutions where CPU and FPGA code are compiled together to create a more efficient, seamless execution. And Amiri et al. [8] propose a workload partitioning strategy, a scheduler that distributes data blocks depending on the throughput of CPU and FPGA. While the works of Rodríguez et al. [99] and Vaishnav et al. [109] present frameworks for resource-elastic scheduling between host and multiple devices, they expect some form of overarching goal between the workloads or the possibility to run the same workload on different computation units (host and/or devices).

Lower-level approaches require the direct handling of the communication between host and device. Generally, this is done by using one of the communication schemata: polling or interrupts. Both are essential concepts for network communications and storage devices, and their trade-offs have been heavily researched. No notification schema is superior to the other in all situations. Instead, hybrid models are used. For example, Dovrolis et al. [32] present the Hybrid Interrupt-Polling model for network interfaces and Salah et al. [101] provide analytical models to analyze the performance of Gigabit-networks. Both works are used for network traffic and arrive at the same conclusion: interrupts for a low network load and polling for a high network load. The explanation for this is that polling with the looping by the user program reserves a fixed amount of CPU load, while system interrupts lead to breaks in the CPU execution. A CPU with a high load would be too slow to process the system interrupts in time, stalling the CPU and significantly deteriorating overall performance. Polling is also the preferred communication schema when using synchronous communication or sending RPC requests. Otherwise, interrupts are preferred, as described by Langendoen et al. [62]. The work by Yang et al. [111] analyzed storage I/O and validated the notification schema selection. System performance improves when using interrupts to transfer large data sizes or when hardware stalls cause long latencies. In contrast, polling improves the system performance when transferring small data sizes or data in high frequency.

As data compression is often part of a pipeline, independent host workloads will likely run concurrently with the compression workload. Thus, it is of paramount importance to fairly and efficiently distribute the system performance on all the workloads, like the policy we present in chapter 5 achieves.

Chapter 3

Background

3.1 CERN

At the European Organization for Nuclear Research (CERN) [19], High Energy Physics (HEP) is studied. Founded in 1954, it is the biggest HEP facility worldwide and the first international organization that considered Germany an equal partner after World War II. Advocating science for peace, the research facility, as shown in Figure 3.1, has one of the largest particle accelerator in the world, called LHC. With a circumference of about 27 km, it accelerates and collides particles at high energy levels. The LHC works together with smaller accelerators to achieve those high energy levels in the particles. Over 16,000 scientists from 110 nations and 76 countries participate in the HEP research based on those collisions [18].

To facilitate state-of-the-art research, CERN regularly undergoes upgrade phases of its infrastructure. The current upgrade *Long Shutdown 2* (LS 2) is a three-year-long major upgrade. This also includes the detectors and infrastructures of the four big LHC experiments ALICE, ATLAS, CMS, and LHCb. After the upgrade, a single collision inside the particle accelerator will create significantly more data.

3.1.1 High Energy Physics

High Energy Physics (HEP) studies the universe's fundamental building blocks and their interactions. This includes, but is not limited to, all types of particles that make up matter and anti-matter and the four natural forces: weak and strong nuclear force, electromagnetic force, and gravitational force. Over the past decades, HEP has created *the Standard Model of Particle Physics* (SM), which attempts to describe all possible interactions between any type of particle. While SM has promising results, it is not complete so far. For example, two significant parts missing are the understanding of how to integrate the gravitational force

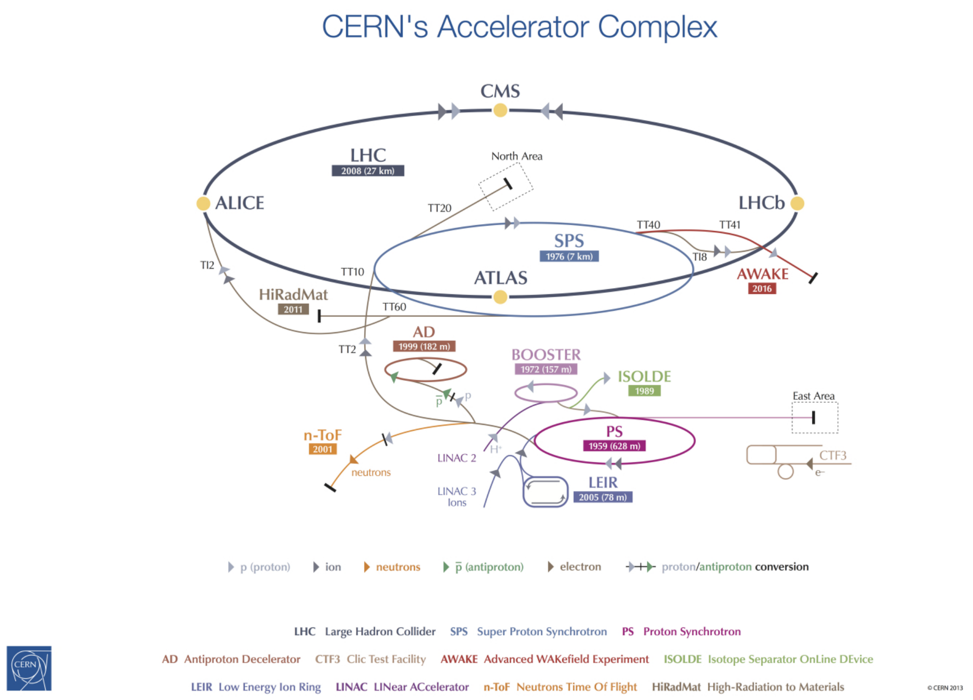


Fig. 3.1 CERN accelerator complex [79]

into the SM and the understanding of the asymmetry between matter and anti-matter, which allows the existence of this universe in the first place.

To gain new insights into HEP, scientists search for rare occurrences of particle interactions that are not fully described by the SM yet. One way of creating this data is by colliding particles at velocities close to the speed of light and detecting the newly created particles. A data acquisition system (DAQ) retrieves, filters, assures the quality and transports the data to permanent storage. The filtering is an essential step as it reduces the data by removing the already well-known particle interactions, leaving the rare occurrences in which the physicists are interested.

3.1.2 Discoveries and Outreach

First and foremost, CERN does fundamental research. Its goal is to improve theoretical models. Applying the models to create technologies and products is not the main focus.

Still, aside from the famous Higgs Boson discovery in 2012 and with this the confirmation of the theoretical proposal from Robert Brout, François Englert and Peter Higgs in the 1960s, which describes how elementary particles get their masses, CERN directly contributes to practical applications for society. For example, CERN's ISOLDE facility provides isotopes for medical applications. And the Medipix3 technology [39] allows taking colored 3D X-rays of humans. This allows identifying not only bones or metal implants but also blood vessels without any use of contrast agents. Another famous contribution was the development of the World Wide Web by Tim Berners-Lee, who worked for CERN at that time. He developed HTML, HTTP, the concept of a URL, and the first web browser. Furthermore, CERN has a knowledge transfer group that works on getting technology developed at CERN applied to the industry.

Another vital part of CERN is outreach. Tours at CERN are open to any visitor free of charge, publications are open access, and special programs exist for school classes.

Overall, even though CERN does fundamental research, it actively works on being beneficial to society through outreach work and knowledge transfer.

3.2 The LHCb Experiment

The LHCb experiment is the smallest of the four large LHC experiments. It has around 1400 members from 86 different universities and laboratories residing in 18 different countries [69].

The LHC-beauty experiment analyzes the decay of b and anti-b quarks to study CP-violation. The CP violation describes a violation of the expected symmetry when a particle is replaced by its antiparticle at its inverted spatial coordinates. One of the CP violation consequences is the so-called matter-antimatter asymmetry. The matter-antimatter asymmetry results in the creation of slightly more matter than antimatter - and as such, allows the existence of our visible universe. B quarks are in particular of interest for this as they are short-lived particles that were common in the aftermath of the Big Bang [68]. The hope is that b quarks will help to understand better why this asymmetry exists.

To capture the b quarks created by the LHC, LHCb has a detector that is 21 m long, 10 m high, 13 m wide, and weighs 5,600 tons. Unlike the other LHC experiments (ALICE, ATLAS, CMS), the detector is not built like an onion around the collision point to capture 360°of the collision. Instead, it is built like a classic fixed-target experiment where collisions are captured in *front* of it, like a giant photo-camera. Due to the nature of b quarks, when particles collide, they are only created in a cone-shaped form in the direction of the original trajectory of the particles. Furthermore, due to the overall majority of symmetry of b quark and anti-b quark, only one-half of the collision needs to be measured. As such, the LHCb experiment has a cost-efficient detector that consists of multiple subdetectors that are placed one after the other. The space saved by this approach allows for comfortable maintenance as

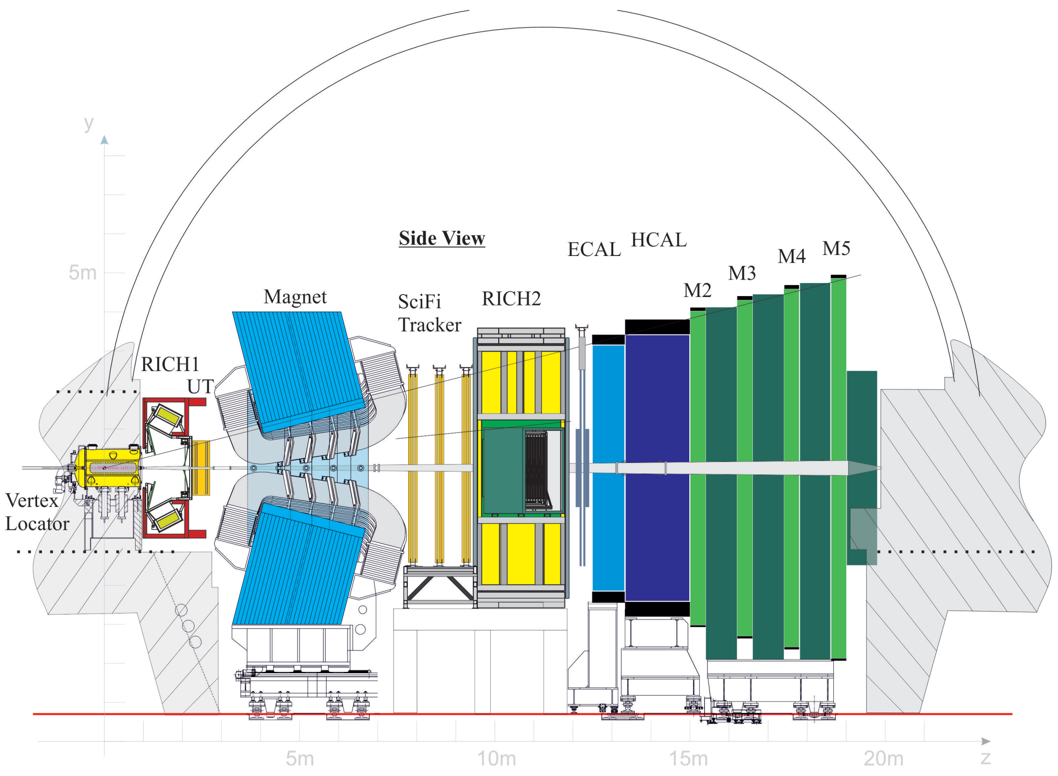


Fig. 3.2 LHCb Detector for Run 3 [70]

it is possible to bring single modules to the surface - something nearly impossible for the onion-shaped detectors. The layout of the detector is shown in Figure 3.2.

The goal of such HEP detectors is to identify particles created by the collision. For this, they consist of three major parts: a magnet to give charged particles a curved (corkscrew) trajectory, some detectors to detect the trajectory, and some detectors that measure the particle's momentum. Based on this, the mass of the particles can be calculated so that the particle can be identified.

The Upgrade: LS2

After the upgrade, for Run 3, the detector will create data at a rate of 4-5 TB/s. For filtering and reducing the data created by the detector, so-called triggers are used. Contrary to the previous run, the DAQ will no longer include a hardware-based trigger. Instead, it will consist of two software-based triggers, High-Level Trigger 1 (HLT1) and High-Level Trigger 2 (HLT2). HLT1 is a real-time *online* trigger and HLT2 is not - it performs *offline*. After the processing by the triggers, only about 0.3% of the detector data is left and stored long-term for physics analyses.

For HLT1, the upgrade means a practical data rate increase by a factor of 67 – 83. To reach this goal, HLT1 is implemented using GPUs. A temporary *HLT1 buffer* between HLT1 and HLT2 is used for performing real-time adjustments of the detector alignment to increase the measurement accuracy and also for decoupling the real-time HLT1 from HLT2. This will now be more significant as the throughput requirements increase by a multiple compared to Run 2. HLT1 will write with 150 GB/s to the HLT1 buffer. In theory, HLT2 should also read with 150 GB/s at the same time. However, the idea is that data is temporarily stored in the buffer, and HLT2 can *catch up* when the LHC has its downtime. More details about the HLT1 buffer, its setup, and limitations, are described in section 7.1.1 as part of the introduction for the cost-benefit analysis of utilizing compression techniques for the DAQ.

3.3 Hardware

In this section, all the hardware used throughout the work is presented. Table 3.1 lists all the servers and the chapters in which they are used. And Table 3.2 lists the compression accelerators we had physically on-site. Additional accelerators we acquired performance data on are the Zipline accelerator [98] resulting out of a collaboration of Broadcom and Microsoft and the NoLoad Computational Storage Processor of Eideticom Communications Inc. [50].

3.4 Data

Table 3.3 contains the descriptions of all the data sets used throughout this study. Data sets derived from one of the data listed here are excluded. In particular, this concerns the LHCb data sets. For chapter 6, each subdetector is extracted to create a separate data set. Afterwards, only the VELO subdetector data is used to evaluate compression techniques further.

Why a chunk size of 157 MB?

Introducing multi-threading to the compression algorithms is only straightforward when considering each unit of compression stream as independent of each other. To efficiently use a system, it would be ideal if each thread had its own compression stream. As such, each compression stream needs its own local data and variables. The largest data set is nearly 5 GB large. This would mean that each thread must have over 10 GB available for input and output buffer of the compression function. On a system like the ARM, which has 256 virtual

Table 3.1 Servers

	Xeon	ThunderX2	POWER8	EPYC	EPYC	*Storage* Xeon	*HLT2* EPYC
Architecture	Intel	ARM	IBM	AMD	AMD	Intel	AMD
Platform	x86_64	aarch64	ppc64le	x86_64	x86_64	x86_64	x86_64
CPU Type	E5-2650 v3 @ 2.30GHz	CN9980 v2.1 @ 2.20GHz	8335-GTB	EPYC 7742 (Rome) @ 2.25 GHz	EPYC 7502 @ 2.5 GHz	Gold 6342 @ 2.80GHz	EPYC 7302 @ 3.0 GHz
Release Date	2014	2018	2016	2019	2019	2021	2019
TDP [W]	105	180	190	225	180	230	155
Virtual Cores	40	256	128	256	64	96	64
Threads per Real Core	2	4	8	2	1	2	2
Real Cores per Socket	10	32	8	64	32	24	16
Sockets	2	2	2	2	2	2	2
NUMA Nodes	2	2	2	8	2	2	2
RAM [GB]	64	256	256	256	512	256	480
Memory Channels per Socket	4	8	8	8	8	8	8
Max. Memory Bandwidth [GB/s]	68	158	230	95	204.8	204.8	204.8
PCIe	Gen3	Gen3	Gen3	Gen4	Gen4	Gen4	Gen4
Used in Chapters	4	4	4	5	6	6, 7	7

Table 3.2 On-Site Compression Accelerators

Manufacturer	AHA Product Groups	Intel
Model	AHA378 [44]	QAT8970[27]
Compression Algorithms	lzs, deflate	deflate
Data Formats	raw, gzip, zlib	raw, zlib
Compression Levels	1	4
PCIe card	Gen3 x16	Gen3
Height	half-size	half-size
Width	double-width	double-width
Technology	FPGA	ASIC
Miscellaneous	-	Supports Encryption
Advertised Throughput [Gb/s]	80	66
Power Consumption [W]	60	23

cores, 2.5 TB RAM would be needed. Even when only using real cores, it would still need more than 625 GB RAM. This is not feasible.

Thus, a study was performed to evaluate if a smaller chunk of the data set can be obtained with a similar compression ratio and average compression speed as the entire data set. Figure 3.3 shows this for the two data sets, ATLAS 1 and LHCb 1. It plots the compression and throughput performance when reducing the chunk size. Starting from a chunk size of 157 MB and larger, a similar performance behavior compared to the entire data set can be seen. This is marked as a vertical orange line. The 157 MB mark does not mean that everything is discarded after the first 157 MB of a data set. It only means that each compression stream compresses an input size of 157 MB (*chunk size*) one at a time. The entire data set will be traversed using round-robin to extract 157 MB large data chunks for the compression streams that run in parallel.

This evaluation was done using the single-threaded version of `CPUComp`.

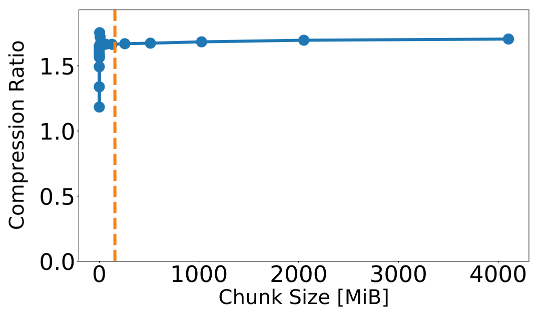
3.5 Software Tools

3.5.1 Single-Threaded CPUComp

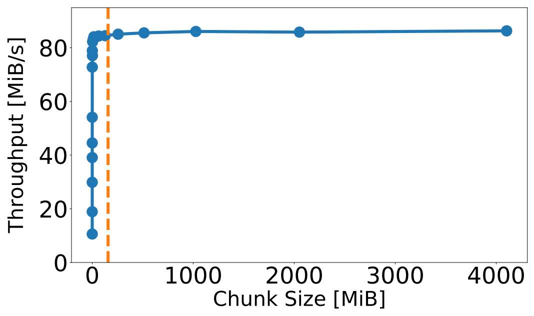
`CPUComp` is a single-threaded program that allows to utilize and benchmark multiple popular general-purpose lossless compression algorithms. They are `bzip2`, `lz4`, `lz4_HC`, `snappy`, `zlib`, `zstd` and `xz`. If available, their compression level can be selected. It allows to compress an entire input file and optionally writes the compressed output to a file. Where possible, the output is used to verify the compression implementation using the default command-line tool

Table 3.3 Data sets

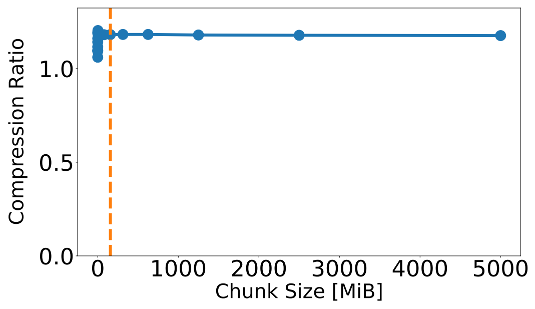
Name	File Size	Content	Compression Ratio zlib level 2	Miscellaneous
		Compression Corpora		
enwik9 [73]	1 GB	Extract of the English Wikipedia (2006)	2.9	
proteins.200MB [37]	200 MB	Protein data (Pizza & Chili Corpus)	2.14	
Silesia corpus [4]	203 MB	Files tarred to single file	2.96	
Calgary corpus [4]	3.1 MB	Tarred into a single file and duplicated to reach about 157 MB	2.87	One of the most famous compression corpora
		HEP Data Sets		
ALICE 1	2.1 GB	Lead-Lead Collisions	1.18	Already compressed: Huffman-encoded TPC clusters
ALICE 2	983 MB	Lead-Lead Collisions	1.42	-
ALICE 3	1.9 GB	Proton-Proton Collisions	1.1	Already compressed: Huffman-encoded TPC clusters
ATLAS 1	4.0 GB	Proton-Proton Collisions	1.62	-
ATLAS 2	4.1 GB	Proton-Proton Collisions	1.62	-
LHCb 1	4.9 GB	Full Stream, Proton-Proton Collisions	1.16	2016/17?
LHCb 2	4.6 GB	Full Stream, Proton-Proton Collisions	1.14	2017
LHCb 3	4.0 GB	Full Stream, Proton-Proton Collisions	1.17	2018
CMS 1	5.9 MB	Test data, mainly zeros	4.42	



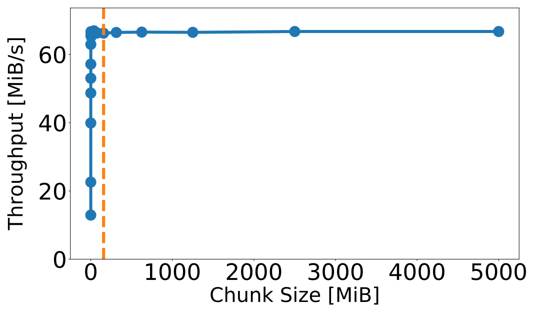
(a) ATLAS 1



(b) ATLAS 1



(c) LHCb 1



(d) LHCb 1

Fig. 3.3 Compression and throughput performance of different chunk sizes. The orange vertical line marks the chunk size of 157 MB.

for the specific algorithm. In some cases, it is not possible as the compression data is written *raw* and without the headers expected by the decompression tool.

The benchmark component of `CPUComp` allows compression tasks to be repeated until a set amount of iterations or a time limit is reached. Afterwards, the statistic is returned as CSV output. Contrary to the later developed multi-threaded version `mtCPUComp`, `CPUComp` is simpler and can only compress the entire data set. `MtCPUComp` will allow simulating *streaming compression*. It can compress smaller data chunks that are extracted from the data set while traversing it.

3.5.2 Non-Uniform Memory Access (NUMA)

Non-Uniform Memory Access (NUMA) describes a design concept for multiprocessing systems. It clusters memory regions around processors, so-called NUMA domains, so that there is a local memory region for each processor with *fast* low latency access. Access to other memory regions will have higher latency. In a smaller multiprocessing system, the access latency to the regions outside the local region will be uniform. In larger multiprocessing systems, this is not necessarily given. Those can have multiple step-ups in latency depending on the physical distance between the processor and the memory regions. Consequently, it is advantageous if it is possible to avoid unnecessary cross-NUMA domain accesses.

In particular, for Linux, two tools allow binding software to NUMA domains, CPU cores, and memory regions. They are `numactl` [9] and `hwloc` [87] (specifically: `hwloc-bind`). Both of them provide a latency matrix describing the latency between the different NUMA domains. While the tools provide nearly the same functionality, the interface is different.

In this work, we use `numactl` to NUMA-bind a program at runtime as the command line commands are more intuitive than `hwloc`. `Hwloc` we use for integrating NUMA-binding within the source code in C/C++ programs. The usage of `hwloc` is necessary as we will design software that needs more complex NUMA binding, and it was impossible to find a C/C++ API to use `numactl` from the source code. For example, the software will bind memory locations to NUMA domains and then only shares those with the threads residing on the same NUMA domain. This is impossible to do with a command-line tool at runtime.

3.5.3 Hardware Performance Counters

Hardware Performance Counters are internal counters provided by the CPU manufacturer to monitor the performance of the CPU. They allow monitoring metrics like power consumption, number of instructions being executed, memory bandwidth, or cache misses. In general, the CPU will only provide a few registers to count the metrics. If more metrics want to be measured, that program needs to be run multiple times. Depending on the tooling being used (e.g., IPMI), metrics for the entire system can also be available. In this work we use four different tools to retrieve the performance counters: AMD μProf [7], Intel Performance Counter Monitor [28], IPMItool [64] and likwid [100].

3.6 Information Theory - Mathematical Background

3.6.1 Information Theory and Compression

Claude Elwood Shannon, an electrical engineer at Bell Labs, published 1948 "A Mathematical Theory of Communication" [104]. With this paper, the field of *Information Theory* was born. Shannon describes a metric how to measure the amount of unique self-information a message contains. The average self-information of a distribution is called *entropy* and for a source S where each symbol has a usage frequency $P(x_i)$ is calculated the follows

$$H(S) = -\sum_{i=1}^{n} P(x_i) log P(x_i)$$

An event with a low frequency contains a lot of self-information, while an event occurring often contains little self-information. This behavior can be modelled by log, as $log(1) = 0$ and $-log(x)$ increases for smaller x within the interval (0, 1]. Shannon further shows that the results of this metric can be directly transferred to binary representation: the entropy is the average amount of bits needed to represent the information.

When having a source S with an alphabet that consists of n-many $symbols$, compression describes the process of changing an alphabet to perform closer to the entropy of a message. The new alphabet is called $code$ and a single element is called $codeword$. In the case of lossy compression, the entropy limit of the source is subverted. Compression techniques can be categorized into several categories

- **Lossless vs lossy compression** — Lossless compression does not go below the limit of entropy, while lossy does.
- **Variable-length code vs fixed-length code** — Defines if all code words are the same length or can vary in length.
- **Instantaneous Code** – The last element of a code word signifies the end of this code word. The first element of the next code word does not need to be read to realize the end of the previous code word.
- **Prefix Code** – No shorter code word is the prefix of a longer code word; as such, each code word can be uniquely identified (Kraft–McMillan inequality).

3.6.2 Lossy Compression

Lossy compression has to evaluate two metrics: compression ratio and rate distortion. Rate distortion describes how much noise is introduced when compressing a source. To evaluate the rate distortion, multiple metrics can be used. They include the mean square error (MSE) and peak-signal to noise ratio (PSNR) [see Chapter 6].

3.6.3 Compression Techniques

The majority of compression techniques belong to a certain class of compression algorithms that share similar features. Here is a non-exhaustive list of such classifications.

- **Entropy Coding** — Variable Length Code where more frequent symbols get shorter codewords (e.g., Huffman coding)

- **Dictionary Coding** — A dictionary of symbols where the index of the entry is encoded (e.g., digram coding), can be adaptive (e.g., LZ77/78)
- **Predictive Coding** — Context-based coding, often consists of prediction for chains of symbols (e.g., Prediction with Partial Matching [PPM], Residual Coding)
- **Quantization** — Lossy method to encode ranges of symbols into a single codeword, can be scalar or vector
- **Transform Coding** — Transform data into a different space where the majority of information is focused within the first dimensions. Higher dimensions can be discarded (e.g., Fourier Transformation, Discrete Cosine Transformation, Principal Component Analysis, Wavelets)

3.6.4 Compression Software Libraries

This section contains descriptions of all general-purpose lossless compression libraries used in this work.

Bzip2

Bzip2 [55] is an open-source compression library released by Seward in 1996 under a BSD-style license. It consists of three steps: 1) Burrows-Wheeler algorithm, 2) Move-To-Front transform, and 3) Huffman coding. Contrary to the other algorithms, the compression level does not change the compression method being used but the block size of the input data (between 100 and 900 kB).

Lz4

Lz4 [72] is a fast, low compressing, LZ77 byte-oriented algorithm released by Yann Collet in 2011. Its focus is to have an *extremely* fast decoder. Lz4 has two versions: *lz4* and *lz4_HC*. Lz4_HC offers a compression ratio increase but with a significant speed reduction. Only Lz4_HC is of interest for our work as lz4 offers only a minimal compression ratio for our HEP data. The library of lz4 is distributed under the BSD licenses, while everything else, including the test and example programs, is distributed under the GPLv2 license.

Snappy

Similar to lz4, snappy [43] is a fast, low compressing LZ77 variant. Also, like lz4, the decompression is significantly faster than the compression. It was designed by Google and released in 2011 under a BSD-style license. Snappy has only one compression level.

Xz

Xz Utils [24] is a well-known compression tool developed within the Tukaani Project in 2005. It consists of two components: the command-line tool *xz* and the software library *liblzma*. Xz is most famous for its LZMA2 compression algorithm. LZMA2 is a high compressing, very slow algorithm. It is the default algorithm for the *xz* command-line tool. Most parts of XZ Utils, including liblzma, are in the public domain, while other parts are under different free software licenses such as GNU LGPLv2.1, GNU GPLv2, or GNU GPLv3.

Zlib

Zlib [5] is a compression library written by Jean-Loup Gailly and Mark Adler and released in 1995. It uses the deflate [30] algorithm. Deflate uses a variation of LZ77 and is then followed by Huffman coding to compress data. It is the de facto industry standard of compression libraries. Deflate is used in many programs and operating systems and has many hardware implementations. Zlib has its own open-source license where the origin of the software must not be misrepresented, altered code must be clearly marked, and it is not allowed to remove the license from any source distribution.

Zstd

Zstandard [115], short *zstd*, was designed to compress data in real-time. Amongst others, it was developed by the lz4 inventor Yann Collet as his work at Facebook and it was released in 2015. It is also an LZ77-based algorithm. In addition, it uses fast Finite State Entropy and Huffman coding. It offers 23 different compression levels. Levels above 20 use different compression methods that result in a better compression ratio but also use significantly more memory. It is published under BSD and GPLv2 licenses.

Chapter 4

Performance Characteristics Across CPU Architectures

4.1 Introduction

Data compression tends to become a critical part of the system as soon as it is utilized because it is generally applied to data vital to the process. Therefore, the reliability of the information reconstruction after compression and decompression must be given. As such, utilizing well-tested *standard* compression algorithms is often preferred as it gives a higher degree of confidence to reconstruct the compressed data also decades later flawlessly.

At the same time, we showed in chapter 2: Related Work that research is very limited when it comes to comparing the performance of such algorithms. The limitations range from selecting too few algorithms to too few or a non-diverse selection of data sets (mainly text-based) to comparing those in the majority of cases only to exotic hardware, but not in between different CPU platforms.

Thus, this chapter analyzes the cross-(CPU) platform performance of well-established general-purpose lossless compression algorithms. This will allow us to evaluate if the choice of compression algorithm has to be considered for the hardware choice as part of the overall system design. The results presented here are an extension of the publication Promberger et al. [90].

Table 4.1 Servers

	Xeon	ThunderX2	POWER8
Architecture	Intel	ARM	IBM
Platform	x86_64	aarch64	ppc64le
CPU Type	E5-2650 v3 @ 2.30GHz	CN9980 v2.1 @ 2.20GHz	8335-GTB
Release Date	2014	2018	2016
TDP [W]	105	180	190
Virtual Cores	40	256	128
Threads per Real Core	2	4	8
Real Cores per Socket	10	32	8
Sockets/NUMA nodes	2	2	2
RAM [GB]	64	256	256
Memory Channels	4	8	8
Max. Memory Bandwidth [GB/s]	68	158	230

4.2 Methodology

4.2.1 Hardware

Three servers are used to characterize the compression performance on different CPU platforms. Each server belongs to a different CPU platform: ARM aarch64, IBM POWER8 ppc64le, and Intel x86_64. All of them are dual-socket servers. ARM is the server with the most recent release year. It is a Cavium ThunderX2, a high-tier server from 2018 and also one of the first server processors produced by ARM. For IBM, a POWER8 8335-GTB is used. It is a high-tier server from 2016. And for Intel, a Haswell E5-2650 v3 is used, a mid-tier server from 2014. This and further technical specifications are listed in Table 4.1.

To improve readability for the further section *ARM* refers to the Cavium ThunderX2, *IBM* refers to the 8335-GTB and *Intel* refers to the E5-2650 v3.

4.2.2 Data

The characterization is based on using multiple HEP data sets from the CERN experiments ALICE, ATLAS, and LHCb, as listed in Table 4.2. The data sets are between 980 MB and 4.9 GB in size. They will be processed in chunks of 157 MB, selected via round-robin, as described in section 3.4.

Table 4.2 HEP Data Sets

#	Content	File Size	Compression
		ALICE	
1	Lead-Lead Collisions	2.1 GB	Huffman-encoded TPC clusters
2	Lead-Lead Collisions	983 MB	Uncompressed
3	Proton-Proton Collisions	1.9 GB	Huffman-encoded TPC clusters
		ATLAS	
1	Proton-Proton Collisions	4.0 GB	Uncompressed
2	Proton-Proton Collisions	4.1 GB	Uncompressed
		LHCb	
1	Proton-Proton Collisions	4.9 GB	Uncompressed

Table 4.3 Compression Algorithms

Library	bzip2	lz4_HC	snappy	xz	zlib	zstd
Level	9	2	-	1, 5	2	2, 21

4.2.3 Compression Algorithms

A total of eight configurations of lossless general-purpose compression algorithms and their levels are selected. The selection is based on usage frequency in other works and perceived state-of-the-art usage in production. The selected compression algorithms and compression levels are listed in Table 4.3 (for description see Section 3.6.4). In the further course, we will refer to lz4_HC as `lz4`.

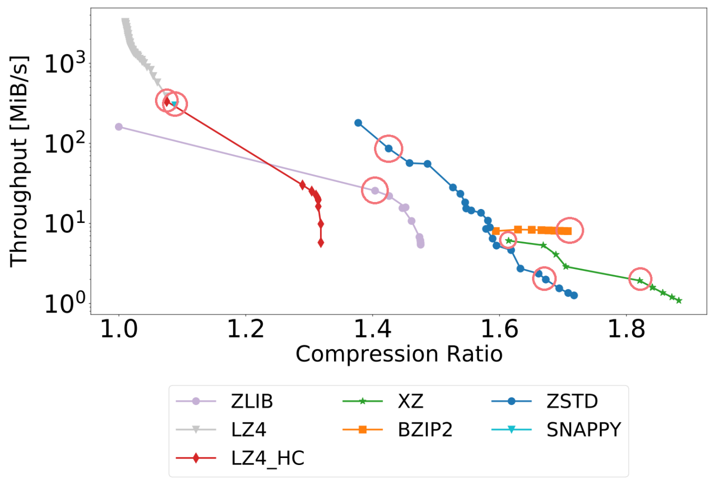
The criterion for selecting the compression levels is based on a pre-study that evaluated the relation between compression ratio and the throughput for each compression level. Shown in Figure 4.1 is the selection of compression level for each algorithm for two data sets, ALICE 1 and LHCb 1. Levels are preferably selected at a “step” where, in relation to the compression ratio, the throughput was still *good* before it significantly degraded.

4.2.4 Software

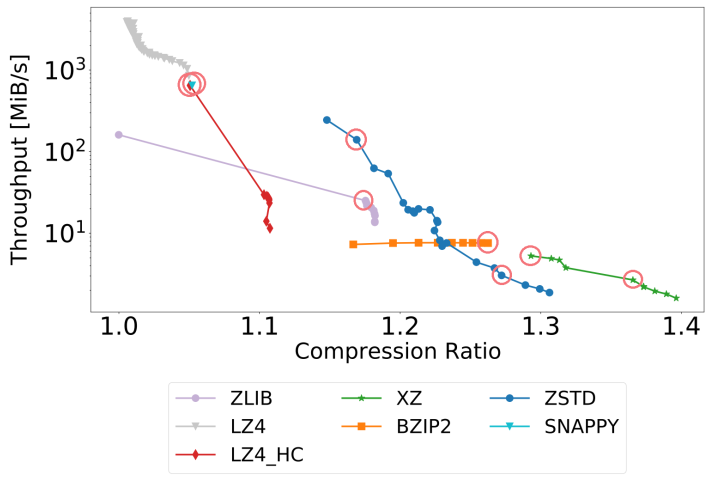
Multi-Threaded CPUComp

A multi-threaded version of `CPUComp` called `mtCPUComp` is developed to evaluate the performance on the different CPU platforms.[1] It allows the selection of compression algorithm, compression level, data set, chunk size for the data set and run time. During the given

[1]The description of single-threaded version can be found in the chapter Background in section 3.5.1



(a) ALICE 2



(b) LHCb 1

Fig. 4.1 Pre-study to evaluate the compression algorithms, shown for the data sets ALICE 2 and LHCb 1. The selected algorithms and their compression levels are circled in red.

run time, data chunks of `chunk size` are created from the data set via round-robin and compressed. The run time is long enough to compensate for warm-up and caching effects. In the end, the output includes the average sustainable compression throughput and average compression ratio. The throughput includes the time reading from memory, compressing, and writing it back to memory. The time to read or write to a hard drive or other devices is not included.

Given that the default compression libraries are single-threaded and HEP data from different collisions are statistically independent data blocks, a straightforward approach for multi-threading is to start a separate thread for each core with each compressing independent chunks of data. As such, each of these `compressThreads` needs its own local memory. For the HEP data sets, a chunk size of at least 157 MB is needed to have a similar compression ratio and compression speed behavior as the entire data set. This means each `compressThread` needs at least 320+ MB of memory per thread for the input and output buffer of the compression.

To optimize the performance and memory, NUMA-binding is used internally within `mtCPUComp` to equally distribute the threads on the NUMA domains and load the entire data set once for each NUMA domain into memory. The memory location is then shared with all threads on the same NUMA domain. Those `compressThreads` then copy the current data chunk into their local input buffer.

Measurement of Power Consumption

The CPU and RAM power consumption per socket is measured in different ways for each server. For Intel, it is obtained with Intel's `Performance Counter Monitor` [28]. For both, ARM and IBM it is obtained using `ipmitool` [64] to access the baseboard management controller. IBM has local access to it, while ARM accesses it via remote-requests. As each CPU manufacturer decides which performance metrics are available, it is impossible to measure the RAM power consumption on the ARM. In addition, only one socket returned proper values for the CPU power consumption (the other is zero). As the load is equally split on both sockets, the ARM power consumption is assumed to be twice the power consumption of the *healthy* reporting socket.

4.3 Results

We look at multiple parameters to characterize the performance of lossless general-purpose compression algorithms. This is especially important as the servers are not very comparable in release year, cost, and performance category (mid-tier vs high-tier). We, therefore, analyze the relative cross-platform performance of the compression algorithms, their normalized performance per core, their robustness depending on data input and algorithm, their scalability, and their energy efficiency.

4.3.1 Cross-Platform Behavior

Cross-platform behavior is an important metric to judge if a compression algorithm prefers a CPU platform (and architecture) over another. This was not the case, as independent of the CPU platform, the relative throughput performance between algorithm `A` and algorithm

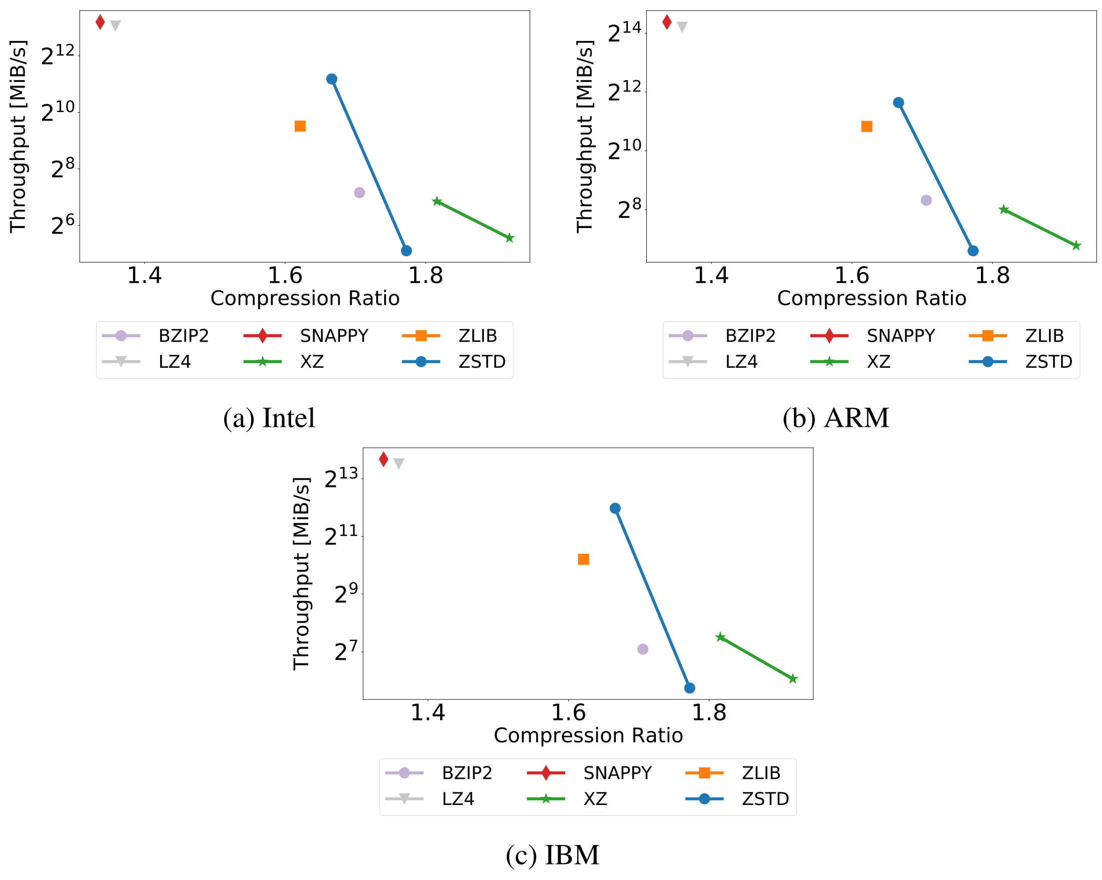
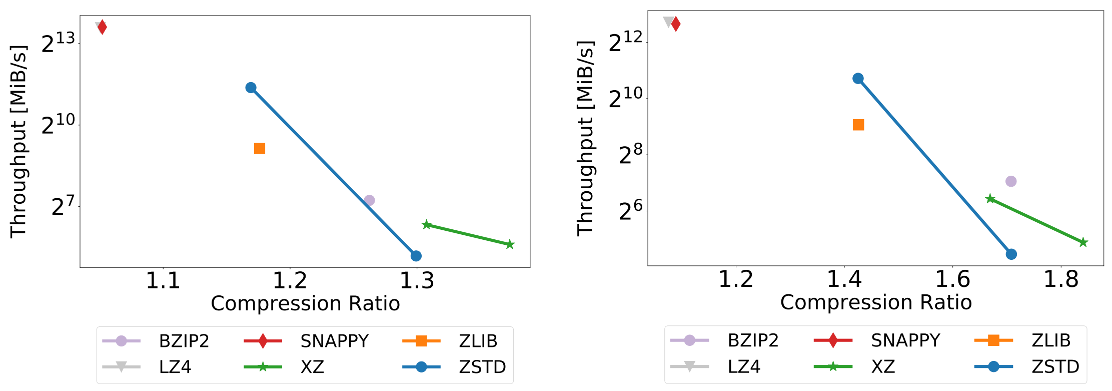


Fig. 4.2 Cross-platform behavior: On all platforms, the algorithms create a similar performance pattern for the same data set (`ATLAS 1` shown).

`B` stays the same. Differences are visible between different data sets, but the relative performance for the algorithms on the same data set is independent of the CPU platform. In a few cases, small changes in the relative order of performance are seen for algorithms with a similar performance, both in compression ratio and throughput. Shown in Figure 4.2 is the performance for ATLAS 1 on the different servers. A slight change in order can be seen on for `xz level 1` and `bzip2`. On IBM the slight throughput differences are switched (`xz level 1` faster than `bzip2`) compared to ARM and Intel. But overall, it shows that ranking compression algorithms by performance (compression ratio, throughput) is platform-independent. While in Figure 4.3, the data sets LHCb 1 and ALICE 2 on Intel show that the ordering of compression algorithms is dependent on the data set. This behavior is seen on all platforms.



(a) Intel - LHCb 1 (b) Intel - ALICE 2

Fig. 4.3 Cross-platform behavior: On the same platform, the algorithms create different performance patterns for different data sets.

4.3.2 Performance Per Core

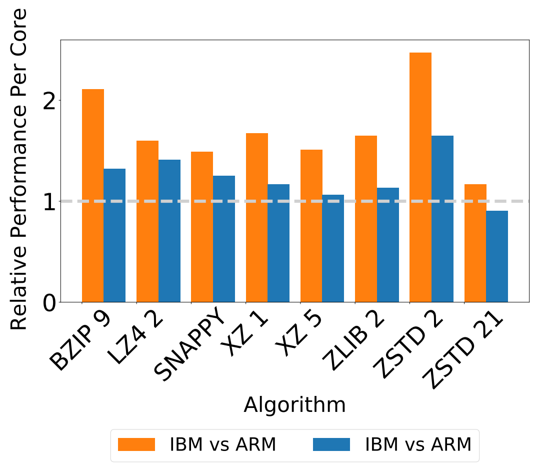
The servers have very different arrangements of number of cores, SMT, clock speed, and TDP. To compare the performance of a single core across platforms, a unifying metric is designed to allow a fair comparison. The metric gives the average performance normalized by clock speed and the number of real cores. The equation is

$$\frac{Throughput}{Clock\ Speed\ *\ \#Real\ Cores}$$

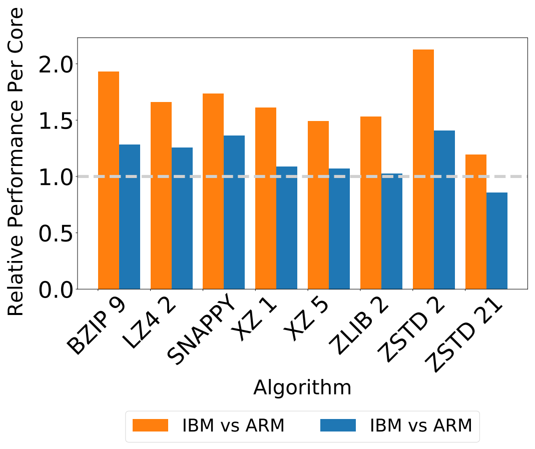
One handicap of this equation is that the clock frequency is not reported during the data taking of this study. Instead, the clock speed defined in the CPU manufacturer specifications is used: ARM 2.2 GHz, IBM 3.3 GHz, and Intel 2.3 GHz. This selected clock speed is the non-turbo one which represents best the performance when utilizing the entire server. Servers usually only use the turbo clock speed for a short duration or low core utilization. Otherwise, they run into power and thermal constraints.

ARM has the weakest per core performance when using the formula, and IBM has the strongest. This performance is the average performance of all data sets. Only for `zstd level 21` ARM performs better. Here, it performs better than Intel but still worse than IBM.

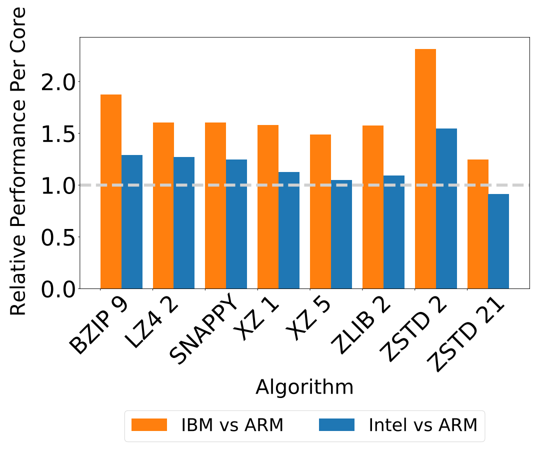
Figure 4.4 shows the factor of performance increase of *performance per core* of IBM and Intel compared to the weakest server ARM. The baseline ARM is shown as a dashed line at the y-axis value of 1.0. Subplot (a) shows the result for data set ALICE 2 and subplot (b) for data set LHCb. Subplot (c) shows it by the algorithm (averaged over data sets), and subplot (d) shows it by the data set (averaged over algorithms)



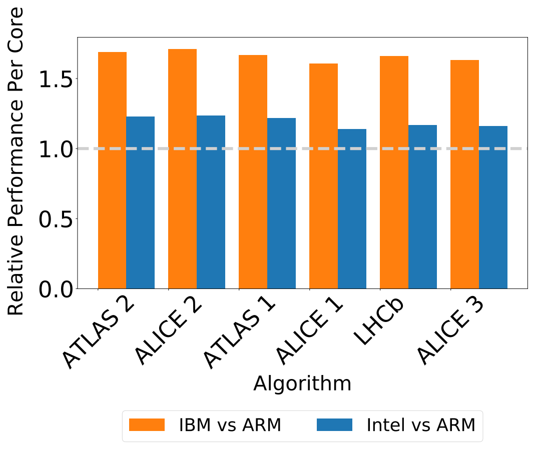
(a) ALICE 2



(b) LHCb



(c) By Algorithm



(d) By Data Set

Fig. 4.4 Performance increase of *performance per core* compared to the weakest server (ARM) shown as dashed line. Subplot (a) shows the data set ALICE 2 and (b) the data set LHCb. Subplot (c) shows it by algorithm (averaged over data sets) and (d) shows it by data set (averaged over algorithms). The orange bars show how much IBM gained in performance compared to ARM, and the blue bars show the performance gain for Intel.

In all cases, the relative performance increase of IBM and Intel compared to ARM behaves on a similar trajectory. This is seen in both cases: when averaging over all data sets and when averaging over all compression algorithms and levels. The data sets only have a marginal influence on the performance characteristics. `Zstd level 2` always has the biggest gain, while `zstd level 21` has the worst. For `zstd level 21`, ARM is even better than Intel. On average, IBM has a better performance by a factor of 1.66 and Intel by a factor of 1.19 compared to the per-core performance of ARM.

4.3.3 Robustness

In a multi-threaded environment, robustness describes the performance stability in relation to the number of threads utilized while varying input parameters of a multi-threaded software. For `mtCPUComp`, the robustness is evaluated by varying the input parameters: data set, compression algorithm and level while running with a multiple of real cores. This is denoted by `SMTx` with x representing the number of a multiple of real cores, e.g. `SMT1` only utilizes the real cores, while `SMT2` uses 2 threads per core. For IBM only `SMT1`, `SMT4`, `SMT6`, `SMT7`, `SMT8` are evaluated as access to the machine was time-limited. This is thought to be a reasonable reduction as shown in Figure 4.5 the performance showed little variance after `SMT4` (= 64 cores).

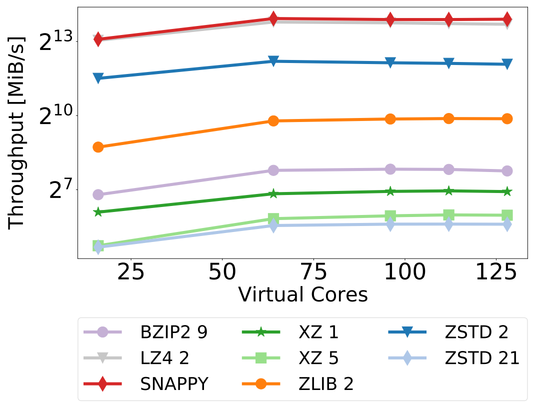
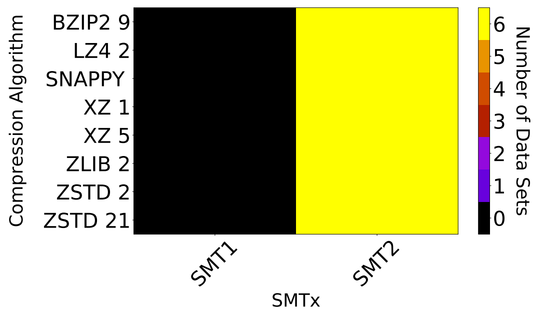


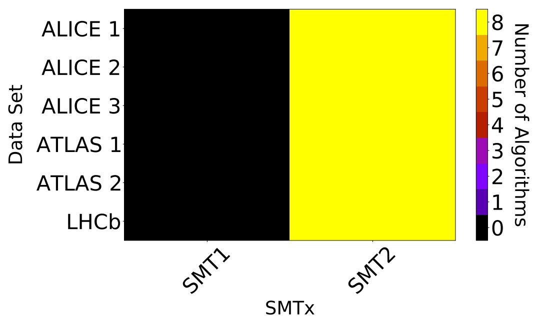
Fig. 4.5 Performance of different compression algorithms for IBM. The performance variance is greatly reduced after `SMT4` (= 64 cores).

The robustness is evaluated in two different scenarios: 1) by compression algorithm (performance is averaged over data sets) and 2) by data set (performance is averaged over compression algorithms and levels). The results are presented as a heatmap in Figure 4.6. The left column shows scenario 1) the robustness by compression algorithm. The heatmap describes how many of the six data sets perform best for a given `SMTx`. The brighter the color, the more data sets prefer that `SMTx` configuration. Yellow represents that all data sets prefer it. The right column shows scenario 2) the robustness by data set. The heatmap here shows how many of the eight compression algorithms and levels have the best performance with the same `SMTx`. To get a better impression of the tendency which `SMTx` is preferred, throughputs within 1% of the maximum throughput of the same input parameters (but different `SMTx`) are considered to be equal.

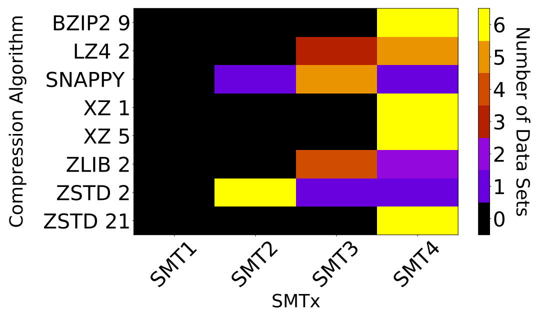
For Intel, in both scenarios, 1) by compression algorithms and 2) by data sets, using all virtual cores achieves, in all cases, the optimal performance. On average, the same also applies to ARM. Some divergence is visible on the heatmap: for scenario 1) `zstd level 21` prefers SMT2 and both, `zlib level 2` and `snappy` prefer SMT3. While for scenario 2) using all virtual cores (SMT4) is the preferred configuration. IBM is the least robust server. Depending on the algorithm, it prefers anything in between SMT4 to SMT8. On average, as shown for scenario 2), the preferred configuration would be SMT7. However, for algorithms `lz4 level 2`, `snappy` and `zstd level 2` the clear preference is SMT4. Using all virtual cores (SMT8) is only a performant option for `xz level 5` and `zlib level 2`.



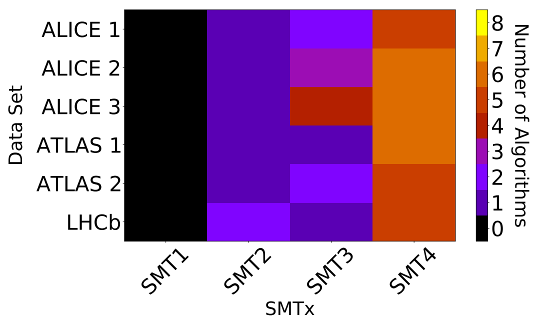
(a) Intel



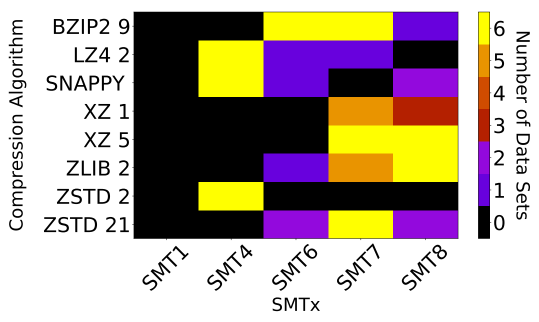
(b) Intel



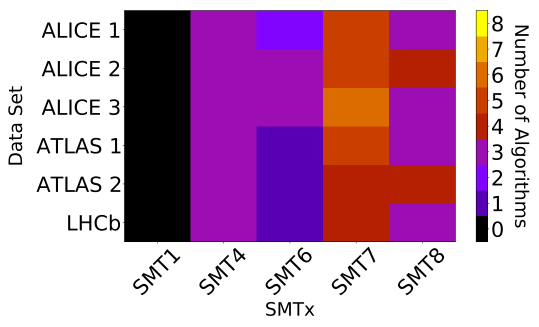
(c) ARM



(d) ARM



(e) IBM



(f) IBM

Fig. 4.6 Robustness shown as a heatmap for the different servers. The left column describes the robustness by compression algorithm, and the right column the robustness by data sets. The brighter the color, the more data points prefer the same SMTx and as such represent the robustness.

Overall, the robustness of a server degrades as the number of available `SMTx` increases. On average, ARM and Intel prefer to use all virtual cores, whereas the preference of IBM is dependent on the input parameters with a preference for `SMT7` and not `SMT8` (all virtual cores).

4.3.4 Scalability

Scalability describes how (in-)efficiently a software performs when run on multiple cores compared to the single-core performance. Utilizing SMT can also help scale more efficiently if the software is not compute-bound. As each of the servers has a different maximal SMTx, an additional question is if the number of SMT is also reflected in the scaling efficiency.

Presented in Table 4.4 are the scalability factors normalized to a single core. The scalability factor is calculated by taking the best multi-threaded `mtCPUComp` performance normalized by the number of real cores and then compared to the single-threaded `CPUComp` performance. The best multi-threaded performance may vary in the number of threads being used depending on the data set and compression algorithm (see subsection 4.3.3).

Our data shows that the scalability is mainly influenced by the choice of algorithm, while the choice of data set has only a minor influence. Averaging over both algorithms and data sets, we achieve the following scaling factors (in the sequence of the maximum SMTx): IBM 1.91, ARM 1.14, and Intel 0.97. Intel is the only server with a mean negative scaling factor, i.e., while one gains a performance increase due to using all virtual cores, each real core performs a bit worse than if it were alone. Most of the algorithms on Intel scaled close to, but just above a factor of 1, while `zstd level 21` is the big negative outlier and `zlib level 2` is the positive outlier. On ARM, `zlib level 2` also performs the best, while `zstd level 2` performs the worst. `Zstd level 2` is also on IBM the worst, while `xz level 5` has the best scaling factor. In reverse order, relative to the SMT available, the best scaling factor only achieves 30% of `SMT8` for IBM, 37% of `SMT4` for ARM and 60% of `SMT2` for Intel.

4.3.5 Power Consumption and Efficiency

This section analyzes the power consumption characteristics. The first part is about the power envelope: the server's power consumption in relation to the TDP and compression performance. And the second part is about the power efficiency of each server.

Power Consumption

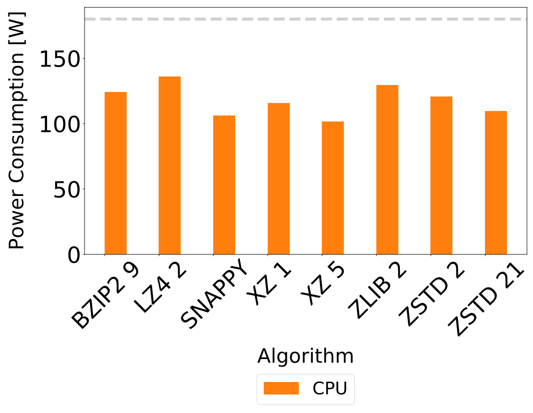
The CPU power consumption depends on the algorithm used. Figure 4.7 shows the CPU power consumption for each server and compression algorithm and level. The average power

Table 4.4 Scalability factor normalized by number of real cores: Best multi-stream compared to single-stream.

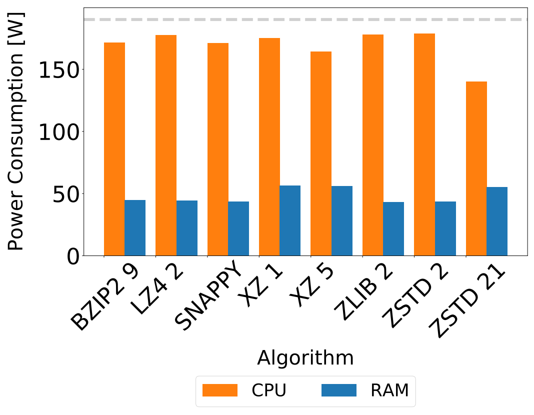
Averaged over Data Sets								
Server	Algorithm							
	bzip2 level 9	lz4 level 2	snappy	xz level 1	xz level 5	zlib level 2	zstd level 2	zstd level 21
ARM	1.14	1.17	1.20	0.98	1.19	1.47 ⇑	0.90 ⇓	1.05
IBM	2.06	1.59	1.68	1.91	2.33 ⇑	2.23	1.58 ⇓	1.87
Intel	1.02	1.01	1.06	0.84	0.99	1.19 ⇑	0.97	0.66 ⇓

Averaged over Algorithms and Experiments								
Server	Experiment							
	ALICE 1	ALICE 2	ALICE 3	ATLAS 1	ATLAS 2	LHCb 1		Average
ARM	1.16	1.11	1.14	1.15	1.15	1.15		1.14
IBM	1.89	1.86	1.92	1.93	1.92	1.92		1.91 ⇑
Intel	0.93	0.97	0.96	0.99	0.99	0.96		0.97 ⇓

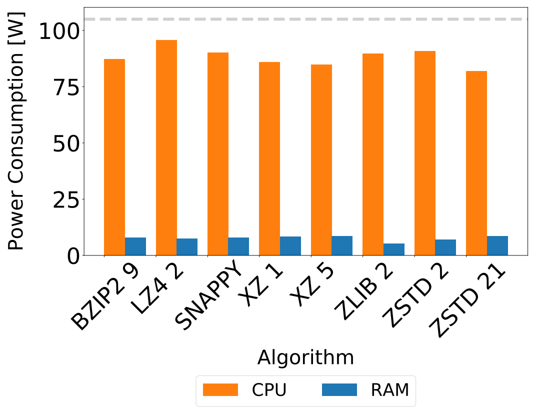
consumption for one socket compared to the TDP is 89.4% for IBM, 83.4% for Intel, and 59.9% for ARM.



(a) ARM



(b) IBM

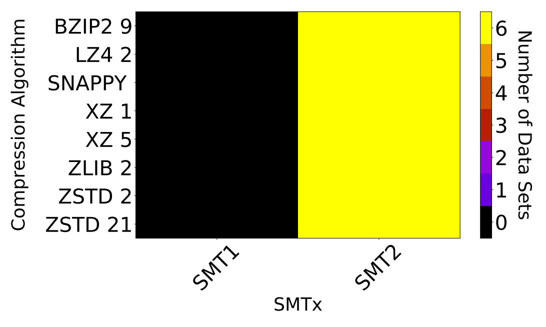


(c) Intel

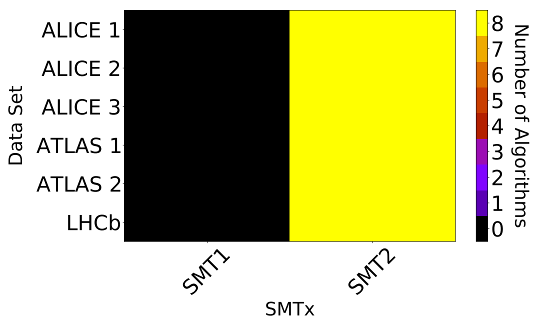
Fig. 4.7 Power consumption for the data set ATLAS 2 for each server and compression algorithm and level: In orange is the CPU power consumption and in blue is the RAM power consumption for Intel and IBM. The gray dashed line shows the TDP.

In all cases, `lz4 level 2` is the highest or one of the highest power consumers. On IBM three candidates, `zlib level 2`, `lz4 level 2` and `zstd level 2`, have a very high power consumption close to the TDP. On Intel `lz4 level 2` is the highest power consumer, also close to the TDP. This is followed by `snappy` and `zstd level 2`. On ARM `lz4 level 2`, while being significantly less close to the TDP, is the highest power consumer, followed by `zlib level 2`. In all cases, `zstd level 21` is the lowest or one of the lowest power consumers. On ARM, there are in addition `snappy` and `xz level 5` which consume the same as `zstd level 21`.

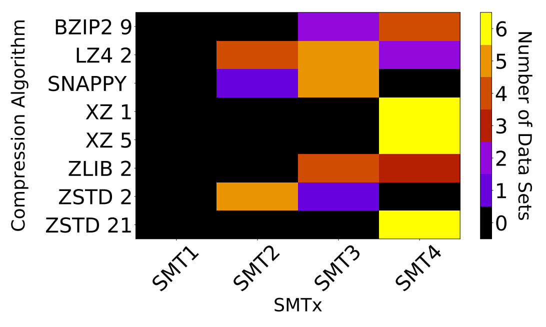
The RAM power consumption is also measured for Intel and IBM. It is impossible to measure it for ARM because the metric is not provided. On Intel, the RAM power consumption is about 8 W per socket with `zlib level 2` using the least (5 W). Combining RAM and CPU power consumption did not change the ranking of which algorithm or level is the smallest or largest power consumer. IBM's RAM consumes on average 50 W per socket. While most algorithms consume around 44 W for the RAM, `xz level 1`, `xz level 5` and `zstd level 21` consume 57 - 60 W. Due to this, `xz level 1` and `xz level 5` become the highest power consumers on the IBM (CPU power consumption only: `zlib level 2`, `lz4 level 2` and `zstd level 2`).



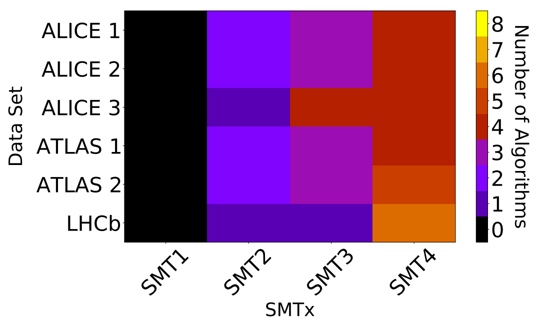
(a) Intel



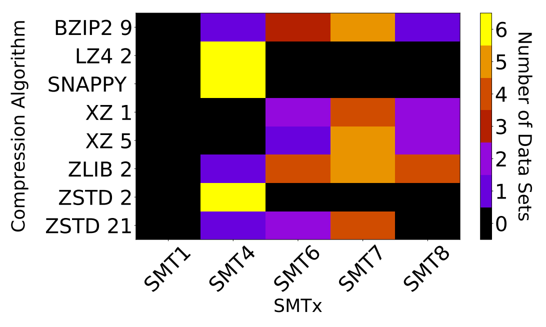
(b) Intel



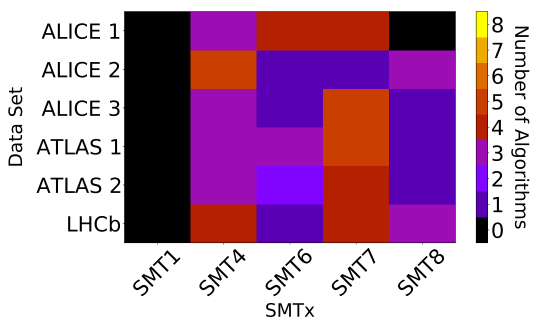
(c) ARM



(d) ARM



(e) IBM

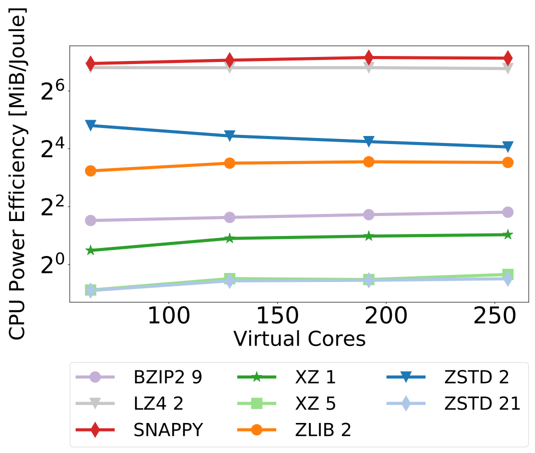


(f) IBM

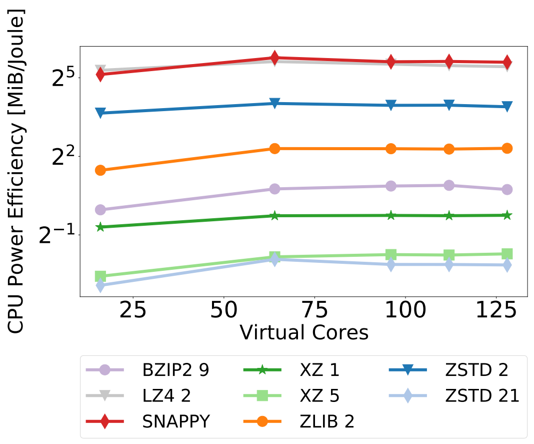
Fig. 4.8 CPU power efficiency shown as heatmap. The left column describes its robustness by compression algorithm, and the right column its robustness by data sets. The brighter the color, the more data points prefer the same `SMTx`.

Power Efficiency

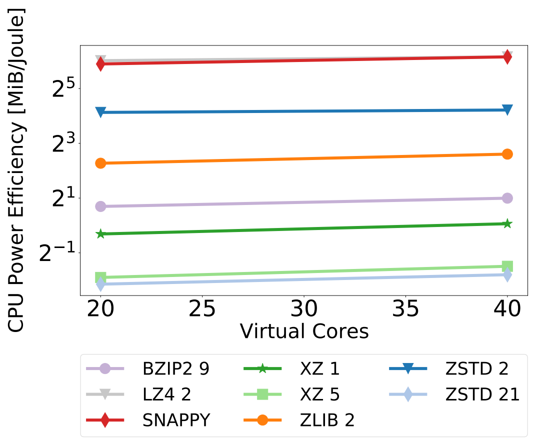
In the context of data compression, power efficiency describes how much data in MiB can be compressed using one Joule of energy. Similar to section 4.3.3 Robustness, Figure 4.8 consists of heatmaps that show the preferred `SMTx` configuration for the two scenarios 1) by compression algorithm and 2) by data set, but this time using the CPU power efficiency as metric (instead of the throughput). Compared to the heatmap in section 4.3.3 Robustness that shows the maximal performance, ARM and IBM are less robust in terms of power efficiency. For scenario 1) ARM became less homogenous, with fewer algorithms preferring `SMT4` and more supporting `SMT3`, while for scenario 2) the results stayed the same with preferring `SMT4`. For IBM, scenario 1) supports very clearly two `SMTx` configurations: `SMT4` for `lz4 level 2`, `snappy` and `zstd level 2` and `SMT7` for the rest, while for scenario 2) the preference stays with `SMT7`.



(a) ARM



(b) IBM



(c) Intel

Fig. 4.9 CPU Power Efficiency of data set ALICE 2

This different behavior can also be seen in Figure 4.9. While overall IBM (subplot b) looks similar to Figure 4.5, which shows the same plot for performance instead of power efficiency, a slight drop in efficiency can be seen, e.g., for `bzip2 level 9` for the maximum `SMTx`. It also shows the unusual power efficiency characteristic of `zstd level 2` on ARM (subplot a) compared to the other CPU platforms and algorithms. It significantly loses efficiency as soon as more than SMT1 (all real cores) is used. The plot again shows the platform independence of these algorithms. On all platforms, the relative order between each algorithm remains the same. `Snappy` and `lz4` being the best performing and `xz level 5` and `zstd level 21` being the worst.

The CPU power efficiency can also be compared across the servers (no RAM power consumption for ARM). In Figure 4.10, the bar chart illustrates the factor of increase in the energy efficiency of ARM and Intel compared to the weakest server (IBM) for each compression algorithm. In all cases, ARM is the most power-efficient server.

`Zstd level 21` is the algorithm for which ARM shows the largest efficiency increase. Here, ARM is 3× better compared to IBM and 2.8× better compared to Intel. The algorithm with the least gain for ARM compared to the other two servers is `zstd level 2`. Here, ARM is only 2.1× better than IBM and 1.8× better than Intel.

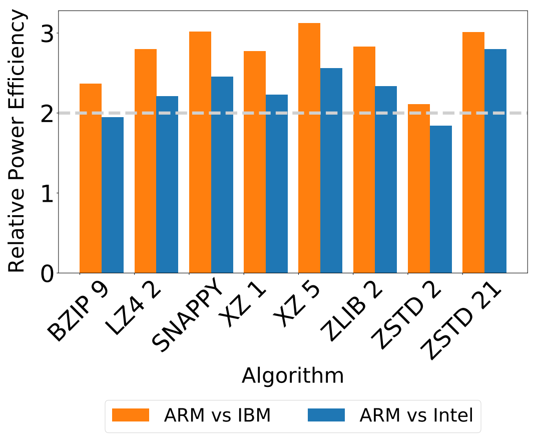


Fig. 4.10 Relative CPU power efficiency of ARM in comparison to Intel and IBM. IBM is the least power efficient server.

4.4 Discussion

This chapter characterized the performance of multiple popular general-purpose lossless compression algorithms on the three common server platforms: ARM aarch64, IBM Power8 ppc64le, and Intel x84_64. We looked at multiple metrics to evaluate if a system design must consider the choice of server platform when data compression using general-purpose lossless compression algorithms is part of the pipeline.

The first metric analyzes the cross-platform behavior. It shows that there is no preference for the CPU platform or architecture when it comes to the relative performance of compression algorithms in terms of compression ratio and throughput. Algorithms A, B, and C, ranked by their performance, would stay in that exact order, independent of the platform being run on. Whereas the input data set influences the ranking of the algorithms when arranged according to their performance. An explanation for it is the target of the standard compression libraries. They are built to produce correctly compressed and decompressed data independent of the platform and offer long-term support. There are custom multi-threaded implementations of those libraries, but they have some downsides. They are not officially maintained, are often limited to a specific platform, are more error-prone due to increased complexity, and do not necessarily guarantee long-term support.

The second metric normalizes the throughput to a unified per core performance to allow a fair comparison between the servers. This is an interesting metric for cost-performance analysis, as the servers from different platforms and performance categories come with varying numbers of cores, frequency, performance, and costs. As such, the sweet spots for acquisition differ. ARM has the worst performance in all cases, and relative to it, IBM has the best. The only exception is `zstd level 21` where Intel performed minimally worse than ARM. The variation of relative performance gain is dependent on the algorithm, with `zstd level 2` having the largest performance gain compared to ARM. It is not dependent on the type of data set.

The third metric, robustness, evaluated how stable the performance is in a multi-threaded environment independent of the input parameters. Or in other terms: how much fine-tuning is needed to select the correct amount of threads to produce optimal performance? A less robust server requires more tests and fine-tuning to gain the best performance and, as such, requires more maintenance and a more careful selection when acquiring the platform. In general, for the selected servers, a correlation existed between a high number of `SMTx` and a less robust performance. Intel, having a maximum of `SMT2`, performs in all cases the best when using all virtual cores. ARM, having a maximum of `SMT4`, on average, performs the best when using all virtual cores. And IBM, having a maximum of `SMT8`, has the least robust environment preferring neither to use only the real cores (`SMT1`) nor to use all virtual cores (`SMT8`), but preferred either `SMT4` or `SMT7`.

Scalability, the fourth metric, evaluated how (in-)efficiently a multi-threaded software performs compared to its single-threaded version. If a software is not compute-bound, increasing the number of `SMTx` results in a better scaling factor. In general, as most software is not fully optimized to be compute-bound, it can be safely assumed that having more `SMTx` available will result in better performing software. While none of the scaling factors measured are close to the maximum `SMTx`, the factors are positively related to the number of `SMTx` available. This means that IBM (max. `SMT8`) achieves with 1.91 the highest scaling factor, ARM (max. `SMT4`) achieves 1.14 and Intel (max. `SMT2`) has a negative scaling factor of 0.97. A negative factor for Intel might have been due to the Turbo setting, which increases the core frequency while staying within safe temperature and power limits when only a low compute load is exerted on the CPU. This means that the single-core performance using only a single real core would perform better than the single-core performance when all virtual cores are utilized. IBM seems to have better utilization of its multi-threading compared to ARM. IBM's factor of 1.91 translates to a 91% performance increase per core for the multi-threaded `mtCPUComp` compared to the single-thread `CPUComp` implementation. ARM only increased its performance by 14%. This means that IBM reached nearly over 6x more performance increase than ARM, even though IBM has only $2\times$ more SMTs than ARM (`SMT8` vs `SMT4`).

With the advent of "green" computing and the increasing cost of electricity, energy efficiency is a factor that cannot be neglected in the system design. Hence, the fifth metric is power consumption. The CPU power consumption is dependent on the algorithms and not the data set. This is expected, as the algorithm decides over the compute intensity by definition. The RAM power consumption is around 7% of TDP for Intel and 26% of TDP for IBM. ARM does not provide the metric. IBM consumes about $6\times$ more power for the RAM than Intel to support $4\times$ the RAM size, $3.4\times$ more memory bandwidth, and $2\times$ more memory channels. While RAM is not part of the TDP, it cannot be neglected for the power envelope of the entire server. Depending on the server, it can be a significant power consumer. The increased RAM power usage for `xz level 1`, `xz level 5` and `zstd level 21`, which are clearly visible on IBM but also visible on a small scale on Intel, might be explained by the high compression ratio achieved by those algorithms. A high compressing algorithm means better pattern recognition, more work on the data set, larger search area, or more complex approaches that translate into more memory accesses and more load on the RAM modules.

Power efficiency relates the software's performance to its power consumption. It is evaluated using the best multi-threaded performance and its respective power consumption. The CPU power efficiency shows that ARM is the most power-efficient server and IBM the least. On average, it is $2.8\times$ more efficient than IBM and $1.3\times$ more efficient than Intel. This underlines that ARM, with its RISC architecture and background in lower-power processors used in embedded or mobile devices, can also create an energy-efficient server CPU. While IBM, which is also a RISC architecture, has a historical background and architectural closeness to mainframes and is not known for being very energy efficient. Mainframes favor very high single-core speed and clock rates as historically, applications were single-threaded. Still nowadays, they are used in banks, insurance companies, and stock exchanges where peak performance, resiliency, and data transfer rates are more critical than a low power consumption [76].

Overall, this study shows that any of the three platforms, ARM, IBM, or Intel, are a viable choice for general-purpose data compression performance. Therefore, the choice of the CPU platform should be based on the hardware preference of the overall system design (e.g., constraints of other software components), the cost-benefit analysis, the cost of increased maintenance required for servers with a high number of SMTs to get the optimal performance, and the energy efficiency.

Designing a new system must fit at least all requirements and, at best, has space to scale in the future. However, how do we integrate data compression into an existing system that was not designed to have it? Data compression is both compute-intense and requires memory bandwidth. Consequently, it is not always possible to integrate general-purpose compression algorithms on the CPU within an acceptable performance degradation of the existing system. In the next chapter, we present one possible solution to this problem: using compression accelerators.

Chapter 5

Efficient Co-scheduling of Heterogeneous Architectures

5.1 Introduction

The previous chapter characterized the performance of lossless general-purpose compression algorithms when executed on different CPU platforms. This is not always possible or the preferred solution. For example, this could be the case when integrating compression in an already optimized system with too few CPU resources to spare.

In this chapter, we will look at one possible solution for this problem: using special compression accelerators and how to optimize the overall system performance to have a robust and fair performance of both host and device (= accelerator) workloads. Accelerators based on FPGAs or ASICs are intensive in development time and cost, making it unattractive on most occasions to develop them for yourself. However, some manufacturers, like Intel [53], the collaboration of Microsoft and Broadcom [98], or AHA [46], offer commercial compression accelerators that allow skipping the implementation effort. In most cases, those accelerators are limited to implementing one or two Lempel-Ziv algorithms, focusing on deflate and lzs. Microsoft offers, in addition, a custom compression algorithm called XP10. The accelerator was developed in a joined project with Microsoft targeting the Microsoft Azure cloud[1].

Heterogeneous systems are inherently complex - including the optimization of performance - even if designed to be heterogeneous in the first place and, as such, allow in-code synergies between host and device workloads. In chapter 2: Related Work, we show that there are many works about frameworks that tightly couple and efficiently schedule host and

[1]At the time of evaluation there was no Linux driver available, yet.

device workloads working towards the same goal. We also show that many works exist for network traffic and storage elements that analyze the low-level communication schemata: polling and interrupts. However, when integrating independent device workloads into an existing system, synergies with host workloads or readily adapted frameworks cannot be expected.

Therefore, this chapter develops a robust, generic policy that optimizes the overall system performance to be fairly shared between host and device workloads. It uses the low-level communication schemata for the device workload, and it relies only on two readily available metrics: power consumption and memory bandwidth. Furthermore, it requires no in-code changes to the CPU workload.

This chapter includes results of the works Promberger et al. [91].

5.2 Methodology

5.2.1 Hardware

Server

The AMD EPYC 7742 is a state-of-the-art dual-socket server. It has 8 memory channels per socket, and each of them has 2 memory slots. However, this particular server has only 4 memory channels on each socket populated, each having a single 32 GB RAM module. With this setup, the theoretical memory bandwidth is 95.37 GiB/s (about 710 Gb/s) per socket. The server has 4 NUMA domains per socket (NPS4). In the BIOS, one can control how the OS sees those NUMA domains: NPS1, NPS2, or NPS4. The accelerator is connected to NUMA node 2 on socket 0 (S0.2). Figure 5.1 shows the setup and Table 5.1 the technical specifications.

Accelerator

The commercial accelerator AHA378 [44] from AHA Products Group is available for this study. It is a half-size, double-width PCIe Gen3 x16 card solely powered by the PCIe slot.

The AHA378 consists of 4 FPGAs, and each of them provides a compression throughput of 20 Gb/s, totaling up to 80 Gb/s. For this, the accelerator consumes on average 60 W. Implemented are two compression algorithms: *deflate* and *lzs*. For deflate, in addition to the raw output, the file formats gzip and zlib are available. Lzs compresses in all cases worse than deflate. Thus, only deflate will be used in the further course. Communicating with the

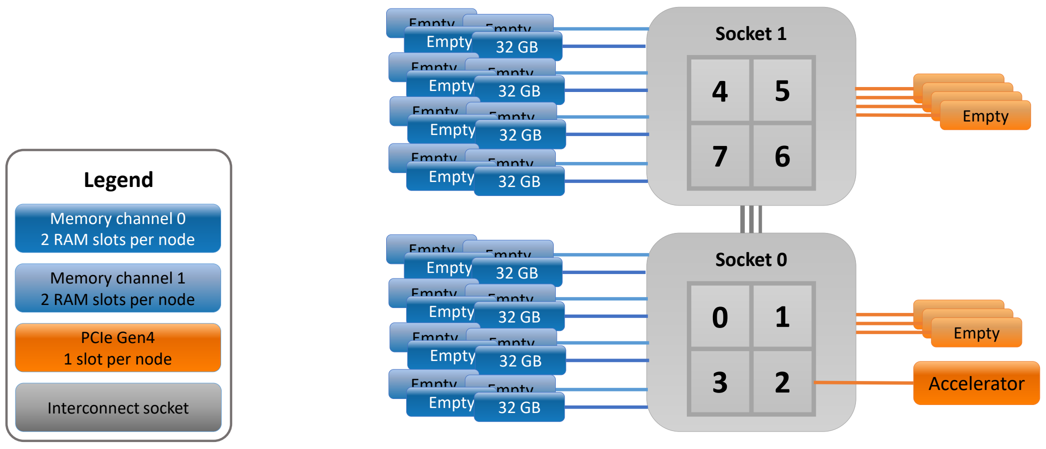


Fig. 5.1 Server Layout. The accelerator is connected to NUMA node S0.2.

AHA378 can either be done with a low-level API or with a zlib-like wrapper. Polling and interrupts are available as communication schemata.

5.2.2 Data

This chapter uses four HEP data sets and four compression corpora. They are listed in Table 5.2. The HEP data sets are one for each of the CERN experiments ALICE, ATLAS, and LHCb, which were also used in the previous chapter. In addition, one data set from CMS is included, which consists of test data and has a very high compression ratio. The four compression corpora are Silesia, Calgary, enwik9, and proteins.200MB of the Pizza & Chili corpus. Silesia and Calgary are collections of different types of files, mainly consisting of text files but also including, e.g., binaries. Enwik9 is the first 1 GB large extract of the English Wikipedia from 2006. And proteins.200MB consists of 200 MB of protein data used by biologists to represent common scientific Big Data. All those corpora, but Calgary corpus, are at least 150 MB large. This selection is taken to be consistent with the size chosen in the previous chapter for the CPU software compression and to be representative of Big Data. Calgary corpus, which is about 3 MB large, is duplicated until reaching a file size of 150 MB. Testing with the CPU compression algorithms from the previous chapter showed that only the high compression ratio algorithms, `xz level 5` and `zstd level 21`, recognized the duplication and achieved a compression ratio of over 160.

Table 5.1 Server

	EPYC
Architecture	AMD
Platform	x86_64
CPU Type	EPYC 7742 (Rome) @ 2.25 GHz
Release Date	2019
TDP [W]	225
Virtual Cores	256
Threads per Real Core	2
Real Cores per Socket	64
Sockets	2
NUMA Nodes	8
RAM [GB]	256
Memory Type	DDR4
Memory Channels per Socket	8
Max. Memory Bandwidth per Socket[GB/s]	95
PCIe	Gen4

Table 5.2 Data Sets

Name	File Size	Content
		Compression Corpora
enwik9 [73]	1 GB	Extract of the English Wikipedia (2006)
proteins.200MB [37]	200 MB	Protein data (Pizza & Chili corpus)
Silesia corpus [4]	203 MB	Files tarred to single file
Calgary corpus [4]	3.1 MB	One of the most famous compression corpora. Tarred into a single file and duplicated to reach about 157 MB
		High Energy Physics Data
ATLAS 1	4.1 GB	Proton-Proton Collision
ALICE 2	1 GB	Lead-Lead Collision, Uncompressed
LHCb 1	4.9 GB	Proton-Proton Collision

5.2.3 Software

Accelerator Software: POLL - Polling

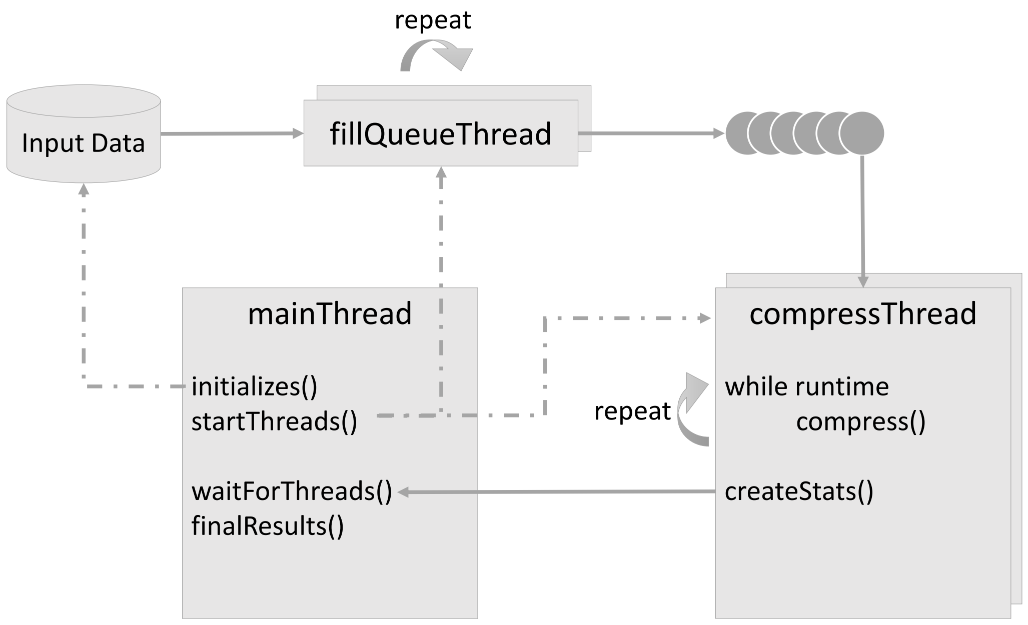


Fig. 5.2 Implementation schematics of POLL

Figure 5.2 shows a schematic of the polling workload POLL. Three types of threads are used: `mainThread`, `fillQueueThread` and `compressThread`. The `mainThread` is started by the user and initializes all shared resources like queues and compression parameters, and it coordinates the other threads. The `fillQueueThreads` create small, 2 MB large data chunks, which are then put in a queue and processed by the `compressThreads`. 2 MB is selected as it consistently achieves the best performance. The data chunks are created via round-robin from the data set to be compressed. The data sets are large enough that the last data chunk can be skipped without problems when smaller than 2 MB. `CompressThreads` are the threads executing the compression. They have their own internal timing and local data, e.g., the output buffer for the compressed data. The uncompressed data chunk is extracted from the queue, which is constantly being filled by `fillQueueThreads`. At the end of the run time, each `compressThread` communicates the results of compression ratio, exact run time, compressed and uncompressed data size to the `mainThread`. It accumulates and presents the final results.

AHA378 needs multiple compression streams within one `compressThread` to get maximum performance. To do this efficiently, a loop loops over all compression streams and acquires the compression status via a non-blocking request. When a compression stream is finished, it is processed.

Accelerator Software: INTRP - Interrupts

The interrupt workload has a similar design to the polling workload, but uses four types of threads: `mainThread`, `submitThread`, `retrieveThread` and `completionThread`. Figure 5.3 shows a schematic of this workload.

The `mainThread` is again started by the user and is responsible for initializing shared resources and coordinating the other threads. It also creates a list of relationships between file descriptors returned by the interrupts and their respective `compressionBlock`. `CompressionBlock` is a custom `struct` which consists of the compression stream and performance statistics like start and end time, compressed and uncompressed data size.

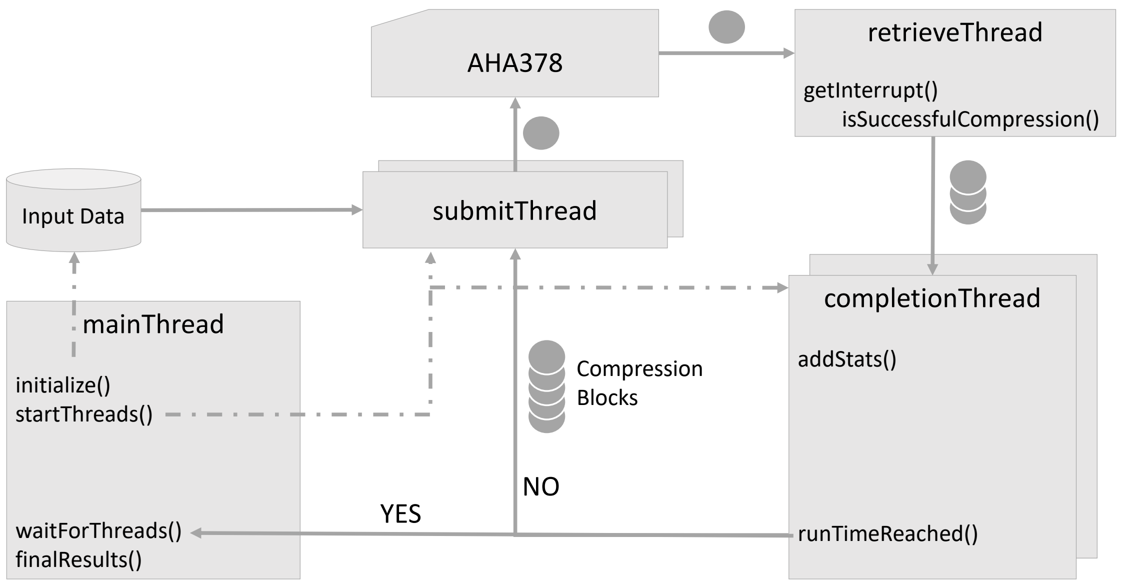


Fig. 5.3 Implementation schematics of INTRP

The `submitThreads` create data chunks of 2 MB from the input data set via round-robin and enqueues it at the accelerator. The `retrieveThread` awaits the accelerator's interrupts. The interrupt signal is then evaluated. If it signifies a successfully executed compression stream, the time and compression size is noted, and it is forwarded to the `completionThreads`. The `completionThreads` handle the accumulation of the results. If the run time is not reached, the `compressionBlock` is reset and enters the queue which the `submitThreads` process.

CPU Software: MLC

The Intel Memory Latency Checker [52] is a tool to measure the memory latency and bandwidth for different access patterns. To get accurate measurements, it has integrated NUMA binding, and it disables the hardware prefetchers. Its measurements are verified with AMD's performance analysis tool AMD μProf [7]. This chapter uses MLC as a memory workload on which the policy is based.

CPU Software: Polynomial

Polynomial is a custom workload based on the work by Choi et al. [22]. They present a roofline model of energy that allows analyzing algorithm performance in relation to the server's power consumption. They provide a polynomial calculation implemented using AVX2 intrinsics and assembly. With this program, they can vary the compute load on the CPU with little load on any other resources, including the memory. We extend the implementation to be multi-threaded and to have a time-based stop criterion instead of a count-based stop criterion. This chapter uses Polynomial as a compute-intensity workload on which the policy is based. It calculates a polynomial of degree 10 on an array of 100,000 double values. The array of doubles is allocated on the stack and shared with all threads to reduce the memory footprint. The throughput is given as *Polynomials calculated per second* (Pol./s).

CPU Software: Reconstruction Software - HLT1

The reconstruction software HLT1 [1] is used by the LHCb experiment as the immediate first step after retrieving the data from the detector and assembling the particle collisions to events. It reconstructs the particles' trajectories to evaluate which particles were created during the collision. It is a fast but coarse filter planned to reduce the data by about a factor of 28. The final implementation will be for GPUs. This CPU implementation is the original implementation on which the GPU version is based. The specific CPU implementation is a benchmark using the build `Moore v51r2` as a basis and the database key `MiniBrunel_2018_MinBias_FTv4_MDF`. It is wrapped in a Python script that supports multi-processing and allows to measure the performance as *Events/s*.

CPU Software: mtCPUComp

`MtCPUComp`, as introduced in the previous chapter in section 4.2.4, is also used in this chapter. The selection of algorithms is reduced to `zlib level 2` and `level 4`, `zstd level 2` and `xz level 5`.

5.3 Policy Overview

In the following sections, we will present our findings that, in the end, result in a co-scheduling policy of (independent) host and compression accelerator workloads. This policy maximizes the system performance while focusing on a fair performance distribution between the workloads. It achieves this by using the optimal NUMA assignment and optimal communication schema for the accelerator. No in-code changes for the host workload are needed. This policy is based on two readily available performance metrics: memory bandwidth and (CPU) power consumption per socket. Server manufacturers will commonly provide tools free of charge to retrieve such metrics, like μProf for AMD.

To arrive at this policy, we will characterize two synthetic workloads, *MLC* and *Polynomial*, which are on opposite ends of the spectrum of resource requirements. MLC requires a high memory bandwidth but puts little pressure on the CPU. Contrary to this, Polynomial, with its calculation of polynomials on the stack, consumes a lot of CPU power while requiring virtually no memory bandwidth. We will characterize their performance: 1) standalone host workloads MLC and Polynomial, 2) standalone device workloads INTRP and POLL, and 3) co-scheduled host workloads with device workloads. This will give us a baseline of expected performance. Based on those findings, we will design a co-scheduling policy. And finally, we will verify our policy using two host workloads, *HLT1* and *mtCPUComp*, that have different performance characteristics.

5.4 Synthetic Benchmarks

5.4.1 The Baseline: Standalone Performance

Table 5.3 lists the results of all the baseline performances of host and device workloads.

Host workloads

MLC, the synthetic memory bandwidth workload, is run in two configurations. One of the configurations is a read-only memory access pattern. As this is the fastest access pattern, it allows us to evaluate how realistic the theoretical peak performance is. The read-only access pattern achieves up to 95% of the theoretical peak performance.

The access pattern of interest for the policy is a 2:1 read/write pattern. It resembles a *copy* access pattern, which was commonly seen in the software we evaluated. With this, a throughput of 1220 Gb/s for the entire server is reached. This is around 86% of the theoretical

Table 5.3 Baseline: Standalone performance. *Slow* and *fast* refer to the data sets with the slowest and fastest throughput.

	Device Workloads		**Host Workloads**	
	POLL	INTRP	MLC	Polynomial
	Throughput [Gb/s]			[Pol./s]
Slow	77.8	80.2	-	-
Fast	82.3	81.6	1220	83.3
Variance	1.7	0.5	-	-
#Threads	27	6	255	254
	Mem. Bandwidth [Gb/s]			
Slow	327	325	-	-
Fast	296	375	1224	42
	Socket CPU Power Consumption [W]			
S0	150 - 151	94 - 98	150	217
S1	83	84	149	215

peak performance. The power consumption is 150 W per socket. This is around 67% of the TDP and just under twice the idle power consumption (83 W).

Polynomial, the synthetic power consumption workload, is run in a single setting described in the methodology section. It reaches a throughput of 83.3 Pol./s. For this, the CPUs consume around 96% of the TDP. The memory bandwidth usage is not even 4% of MLC's memory bandwidth.

Device Workloads

Both device workloads are evaluated using all data sets, and Table 5.3 lists those results.

POLL, the polling workload, achieves the best throughput when utilizing a total of 27 threads: 3 `QueueThreads` and 24 `CompressThreads`. This is about 85% of the socket's real cores. LHCb is the slowest data set which also has the lowest compression ratio. Calgary corpus, Silesia corpus, and enwik9 are the fastest data sets. The average throughput over all data sets is 81.2 Gb/s with a variance of 1.7 Gb/s. The power consumption for S0 is the same as MLC, while S1 is idle.

INTRP, the workload using interrupts, achieves the best throughput when utilizing a total of 7 threads: 1 `FillQueueThread`, 3 `SubmitThreads`, 1 `RetrieveThread` and 2 `CompletionThreads`. This is about 22% of the socket's real cores and nearly 4$\times$ less than POLL needs. LHCb is also the slowest data set, with a slightly higher throughput than for POLL. The best throughput is reached by Calgary corpus, enwik9, and proteins. The average throughput over all data sets is 81.3 Gb/s with a variance of 0.51 Gb/s. This variance is over

Table 5.4 Memory bandwidth and power consumption of idle server. *R/W* stands for read-/write.

Memory Pattern (R/W)	Topology	Bandwidth [Gb/s]
2:1	Node	158
2:1	Socket	630
2:1	Server	1224
1:0	Server	1352
Socket CPU Power Consumption [W]		
Idle	Socket	84
TDP	Socket	225

3× smaller than the variance of POLL. INTRP consumes on S0 only 10 – 15 W more than idle, and S1 is idle.

Table 5.4 lists the memory bandwidth for different topologies and power consumption when the server is idle. The memory bandwidth is acquired using MLC and different read/write access patterns. In theory, 3 of the 4 NUMA nodes should provide enough memory bandwidth for the accelerator. However, all 4 NUMA nodes are needed to achieve maximum performance. Using the memory bandwidth of 3 nodes misses out on at least 10% in performance. Thus, all further characterizations are done using memory-interleaving for NUMA nodes 0-3 on socket S0 for INTRP and POLL.

5.4.2 Co-scheduled Performance

After establishing the baseline for the standalone performance, we characterize the performance when the device workloads, POLL and INTRP, are co-scheduled with the synthetic workloads, MLC and Polynomial. The compression accelerator is connected to node 2 on socket S0, described as S0.2. The device workload uses NUMA binding to bind to the CPU at S0.2 and for the memory to bind to all nodes of socket S0 (S0.0-3).

Table 5.5 lists the performance, memory bandwidth, and power consumption when the host workloads are executed on the same socket S0 as the device workloads. Green and red arrows signify the best and worst throughput, and the pale green row color shows the superior performance comparing POLL and INTRP for the same host workload.

MLC degrades the performance of the device significantly. POLL, which is superior to INTRP, achieves as best throughput of the device at only 45 - 65% (INTRP: 40 - 51%) of the baseline, while MLC reaches 98% of the single-socket performance. Co-scheduled with MLC for both, POLL and INTRP, the power consumption is only marginally higher (5 – 10 W) than MLC standalone consumes. The same holds true when co-scheduled with

Table 5.5 Single socket co-scheduling results of host workloads and device workloads. Accelerator resides on slot S0.2. Listed NUMA-binding only refers to the host workload. Grouped by the host workload, green columns mark superior performance between INTRP and POLL, and the arrows mark the best and worst performance of each workload.

	NUMA binding		Throughput [Gb/s]			Power
					Memory	Consumption [W]
#	Nodes	SMT	Device	Host	Bandwidth	S0
				MLC		
			POLL			
1	2	1	18-19	142-144 ⇓	206-213	-
2	1,2	1	17-18	285-289	348-352	-
3	1,3	1	35-53 ⇑	239-257	386-407	-
4	0,1,3	1	25-38	377-398	491-498	-
5	0,1,3	2	34-50	240-257	383-400	-
6	0,1,2,3	1	16	574-580	623-626	151
7	0,1,2,3	2	5 ⇓	594-596 ⇑	606-609	155-156
			INTRP			
8	2	1	9-10	149-150 ⇓	182-185	-
9	1,2	1	9-10	299-301	332-333	-
10	1,3	1	32-42 ⇑	242-251	385-396	-
11	0,1,3	1	23-31	384-393	491-496	-
12	0,1,3	2	31-42 ⇑	242-251	386-397	-
13	0,1,2,3	1	7-8	601-604	628-630	148-149
14	0,1,2,3	2	3-4 ⇓	605-606 ⇑	610-613	153-155
				Polynomial		
				[Pol./s]		
			POLL			
15	0,1,2,3	1	79-82	32-33	278-312	223-224
16	0,1,2,3	2	80-81 ⇑	36-37 ⇑	277-308	224
			INTRP			
17	0,1,2,3	1	78-82 ⇑	39-40	374-402	224-225
18	0,1,2,3	2	54-73	39-41	361-376	223-224

Polynomial. The NUMA binding of the host workloads creates a similar performance pattern for POLL and INTRP. It is crucial for the device to omit node 2 for MLC. Any configuration that uses node 2 for MLC results for POLL to degrade 77 - 94% in performance and for INTRP to degrade 94 - 96% compared to the baseline. When using nodes 1,3 with SMT1 for MLC or using nodes 0,1,3 with SMT2, the performance is nearly the same. These configurations also achieve the best throughput for the device. Using nodes 0,1,3 with SMT1 degrades the device's performance (down by up to 29%) and supports the performance of MLC. Using SMT2 creates the inverted effect and boosts the performance of the device while reducing the performance of the host workload by the same amount. Overall, the combination MLC and POLL is superior to MLC and INTRP (compare lines #3-5 to #10-12 in Table 5.5). POLL's performance is between 9 - 39% better than INTRP, while the performance of MLC is very similar.

Polynomial, contrary to MLC, has a minimal impact on the device. Even when using all NUMA nodes of the socket, the performance only degrades when INTRP is executed with SMT2. The best performance is achieved using all nodes (S0.0-3), SMT1, and INTRP. The device has a performance loss of $< 3\%$ for slow data sets and no performance loss for the fast data sets, and Polynomial loses a maximum of twice that. For all configurations, the power consumption reaches the TDP.

In the end, the experiments show the importance of the available memory bandwidth for the accelerator and the inferior impact of the power consumption. The best combinations are 1) POLL and MLC and 2) INTRP and Polynomial. This can be explained by 1) POLL reserving a fixed amount of memory bandwidth in a high load scenario of this resource and 2) INTRP making more CPU resources available to the compute-intensive Polynomial.

Aside from these results, we also evaluate the full server performance. They confirm the single socket results of combining POLL with MLC and the marginal impact of Polynomial on the device's performance. Table 5.6 lists an overview of the configurations used to help create the policy. This table also includes the pastel green lines from Table 5.5. They are considered to be the configurations with the *best-combined throughput*, as these configurations balance the performance loss between both host and device workloads. To note is also that Polynomial uses so little memory bandwidth that the full server memory bandwidth does not increase by more than 2% for INTRP, while for POLL, it is between 6 - 17%.

5.5 Policy Design

The development of the policy is motivated by the need to integrate compression accelerators into an existing system that was not planned to run with such devices and where there might

Table 5.6 Results used to design the policy. *Best-combined throughput* is the best fairly balanced performance between host and device throughput. While not explicitly marked, the throughput of Polynomial is given as Pol./s. Grouped by the host workload, green columns mark superior performance between INTRP and POLL.

	Throughput [Gb/s]		Memory	CPU	
Workload	Device	Host	Bandwidth [Gb/s]	Power [W]	#Threads
			Baseline Performance		
POLL	78-82	0	296-327	233	27
INTRP	80-82	0	325-375	178-182	6
MLC	0	1220	1224	299	255
Polynomial	0	83	42	432	254
			Best-Combined Throughput Co-Scheduled - Full Server		Host #Threads
POLL + MLC	63-81	1212-1219	1272	281-290	116
INTRP + MLC	59-78	1248-1253	1272-1273	288	116
POLL + Polynomial	81-82	99-100	326-327	443-445	300,508
INTRP + Polynomial	79-81	100-101	381-414	445	200
			Best-Combined Throughput Co-Scheduled - Single Socket		NUMA Nodes
POLL + MLC	35-53	239-257	386-407	-	(0),1,3
POLL + MLC	25-38	377-398	491-498	-	0,1,3
INTRP + MLC	32-42	242-251	385-396	-	(0),1,3
INTRP + MLC	23-31	384-393	491-496	-	0,1,3
POLL + Polynomial	80-81	36-37	277-308	224	0,1,2,3
INTRP + Polynomial	78-82	39-40	374-402	224-225	0,1,2,3

be no access to the source code of the host workloads. As such, we believe that the policy must fulfill the following requirements

- Improve the overall system performance
- Distribute the performance fairly between device and host workloads
- Reduce the performance loss of the host workloads as much as possible
- Policy parameters should be readily available, independent of the CPU architecture
- No code changes to the host workload

With these requirements in mind, we will design the policy based on our observations in the previous section. Due to not being able to modify the code of all workloads, the performance improvements can only be based on influencing OS scheduling priority or the choice of communication schema for the device.

As parameters on which the policy decisions will be based, we select the two readily available performance metrics: *memory bandwidth* and *power consumption per CPU socket*. Commonly, server manufacturers provide tools free of charge to measure those parameters, making acquiring those metrics architecture-independent. The importance of these parameters can be seen in the previous section, where memory bandwidth significantly influences how the co-scheduling performance will behave. The CPU power consumption can be used as an indicator of the compute intensity. This shortcut can be taken as the characterization results show that MLC, which has a low compute intensity, only consumes 67% of TDP, while Polynomial, which has a high compute intensity, consumes over 95% of TDP.

Therefore, we introduce the following terms to classify the baseline of a host workload and its occupation on the server

- *Compute-heavy*: CPU power needed by the workload is close to the TDP ($> 90\%$)
- *Memory-heavy*: Memory bandwidth of a NUMA node or socket is saturated when combined with the memory bandwidth of the device workload baseline
- *Full server*: Host workload executed on all sockets
- *Separate sockets*: Host workload executed on all sockets, excluding the one on which the device resides

Further, we will design our policy based on limits surrounding memory bandwidth and power consumption. For this, we need the following parameters of server, device, and host workloads

- Server specifications
  - TDP of CPU socket: 225 W
  - Measured memory bandwidth (2:1 read/write)
    - Socket: 630 Gb/s
    - NUMA node: 158 Gb/s
- Baseline performance
  - Device: Required memory bandwidth (max. 375 Gb/s)
  - Host: Required memory bandwidth (best: single socket/node)
  - Host: CPU power consumption

With this knowledge, we designed a policy that uses NUMA-binding and either polling or interrupts to communicate with the device to reach an efficient and fair co-scheduling performance. Our policy is shown in Figure 5.4.

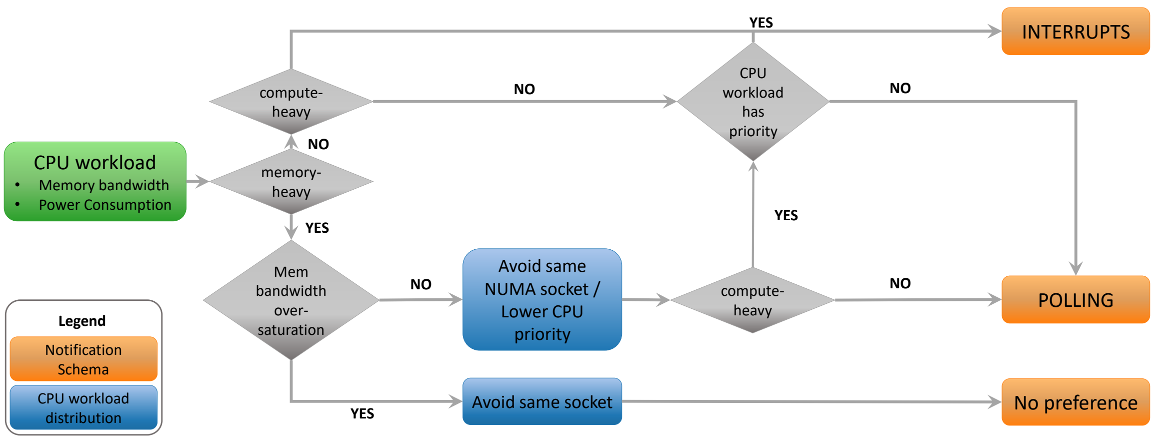


Fig. 5.4 Co-Scheduling Policy

In the previous section, the single socket co-scheduling performance of MLC showed that the occupancy rate of the memory bandwidth is the most critical factor for the device performance. Therefore, it will have the most significant influence on the policy design. The accumulated performance characteristics of host and device baselines can result in four cases that the policy must handle: 1) memory-heavy but not oversaturating the CPU socket, 2) oversaturated memory-heavy, 3) only compute-heavy, and 4) neither. Case 1: the workloads are memory-heavy but do not oversaturate the memory bandwidth of the CPU socket. Then, the host workload can run on the full server but - if possible - should avoid the node to which the accelerator is connected. The choice of communication schema depends on if, in addition, the host workload is also compute-heavy. If yes, use interrupts; otherwise, use polling. Case 2: the workloads greatly oversaturate the memory bandwidth of the CPU socket. Then, the host and device workloads should be run on separate sockets. No preference exists for the communication schema. Case 3: the workloads are not memory-heavy, but the host workload is compute-heavy. Then, the host workload can run on the full server, and the communication schema of choice are interrupts. Case 4: the workloads are neither memory-heavy nor compute-heavy. Then, the host workload can run on the full server, and the choice of communication schema depends on the focus of the system performance. While there will be no significant performance change, using interrupts will slightly prioritize the host workload, and polling will slightly prioritize the device workload.

Independent of the chosen notification schema, the device workload will significantly benefit from 1) NUMA binding, and if a single NUMA node does not support the memory bandwidth, 2) memory-interleaving.

5.6 Policy Verification

We verified the policy using the workloads HLT1 and `mtCPUComp`. `MtCPUComp` is run with the compression algorithms `zlib level 2`, `zstd level 2` and `xz level 5` as they have different profiles of resource usage. The verification is done in three steps: 1) Parameter acquisition, 2) Policy Application, and 3) Evaluation. The evaluation is done using different multi-threaded configurations of the host workload, while the configuration of the device workload is fixed (parameters described in the previous sections).

Step 1: Parameter Acquisition

In the first step, we acquire all necessary parameters. This includes the server performance of memory bandwidth, power consumption, and baseline performance. For server and device, this was done in the previous section. For HLT1, Table 5.7 lists the baseline. The memory bandwidth of HLT1 is for the full server lower than when run on a single socket. Multiple runs confirmed those values.

For `mtCPUComp`, Table 5.8 lists the baseline. To help with the evaluation of `zstd level 2`, both the full server baseline and the single-socket baseline are listed.

Step 2: Policy Application

In the second step, we apply the policy based on the acquired parameters.

Table 5.7 Baseline performance of HLT1.

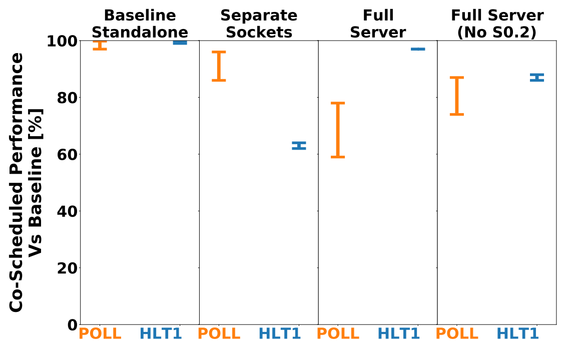
		Single Socket	
	Full Server	S0	S1
Throughput [Event/s]	248887	123957	166768
Mem. Bandwidth [Gb/s]	277	368	328
Power Consumption [W]			
Socket S0	210	218	144
Socket S1	197	94	198
Total	407	312	343
Co-scheduling Choice	Interrupt	Poll (final decision)	

Table 5.8 Baseline performance of `mtCPUComp` for `zlib level 2`, `zstd level 2` and `xz level 5`. For `zstd level 2`, both full socket and single socket performance are given to help with the decision. *Slow* and *fast* refers to the data sets with the slowest and fastest throughput.

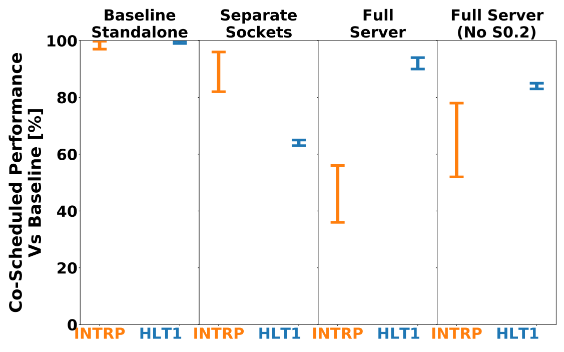
	Full Server		Full Server	Single Socket
	zlib level 2	xz level 5	zstd level 2	zstd level 2
	Throughput [Gb/s]			
Slow	33	2.1	107	66
Fast	74	3.8	156	99
	Memory Bandwidth [Gb/s]			
Slow	120	974	347	290
Fast	158	761	562	426
	Power Consumption [W]			
S0	218-219	178-180	190-208	83
S1	219	180-182	192-215	195-220
Co-scheduling Choice	Interrupt	Poll	Poll? Other Socket?	

The full server performance of HLT1 suggests that HLT1 is compute-heavy. The power consumption is 90% of the TDP, while it only consumes 23% of the server's memory bandwidth. Thus, HLT1 should be using the full server and the communication schema should be interrupts.

However, the single-socket performance comes to a different conclusion. There it is both memory-heavy and compute-heavy. In combination with the compression accelerator's memory bandwidth, it needs about 99 - 118% of the maximum socket's memory bandwidth for a 2:1 read/write access pattern. As such, the communication schema should now be polling, and node S0.2 must be omitted by HLT1. Maybe even running both workloads on separate sockets must be considered.



(a) HLT1 - POLL

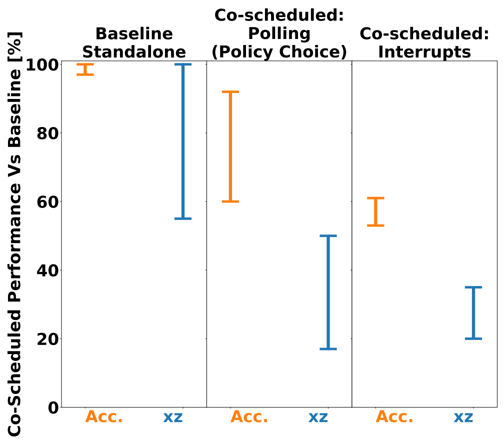


(b) HLT1 - INTRP

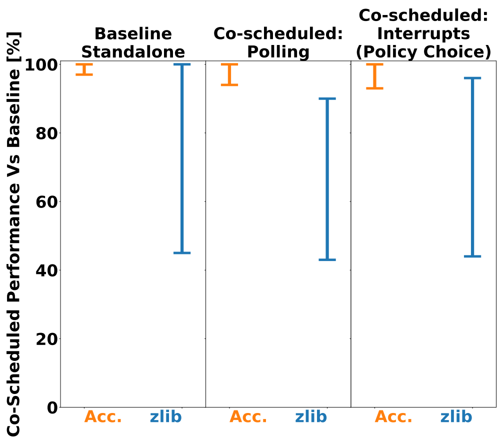
Fig. 5.5 Co-scheduled performance of HLT1 and the accelerator workloads relative to their baseline. As recommended by the policy the best performance is achieved when using POLL and excluding the accelerator's node S0.2 for HLT1.

For `mtCPUComp`, `zlib level 2` and `xz level 5` are clearly identifiable. `Zlib` is compute-heavy, consuming 97% of the TDP and using only 13% of the memory bandwidth. As such, `zlib` should use the full server, and the communication with the device should happen via interrupts. `Xz` is memory-heavy, consuming not more than 81% of the TDP and using 61 - 80% of the full server memory bandwidth. Combining `xz` needs of memory bandwidth and the needs of the compression accelerator will result in a minimal oversaturation of 104 - 110%. As such, `xz` should use the full server but omit node S0.2, and the communication with the device should happen via polling.

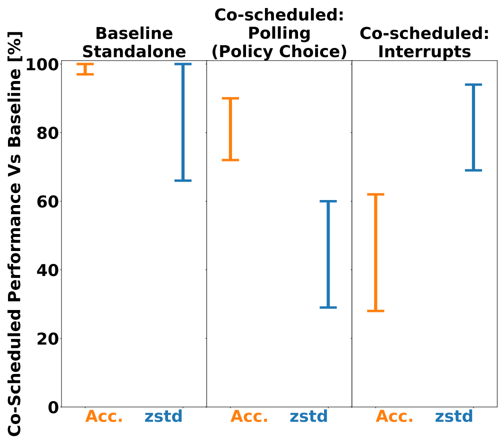
The decision for `zstd level 2` is more complicated, as it is both - memory-heavy and compute-heavy. The policy has two options, either using the full server without node S0.2 and polling or separating the workloads on different sockets. For the full server, the memory bandwidth needed is around 72% for the combined baseline, `zstd level 2` and POLL. While for the single socket, the combined memory bandwidth oversaturates up to 119%. As



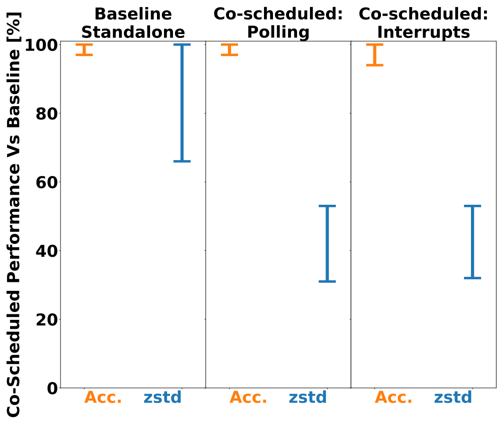
(a) `xz` - Full Server



(b) `zlib` - Full Server



(c) `zstd` - Full Server



(d) `zstd` - Separate Sockets

Fig. 5.6 Co-scheduled best performance of CPUComp `xz`, `zlib`, `zstd` on the full server compared to the baseline for all data sets. For `zstd` the performance on separate sockets is also provided. The results support the policy recommendation which is marked on top of each plot.

such, `zstd level 2` most likely will perform best when using the full server without node S0.2 and using polling to communicate with the compression accelerator. However, as the single-socket performance oversaturates so much, tests should be conducted to verify this.

Step 3: Evaluation

To evaluate the choice of our policy, we co-schedule HLT1 with INTRP and POLL in different configurations (for HLT1): full server with node S0.2, full server without S0.2, and separate sockets. In Figure 5.5 this performance in relation to the baseline is shown. The bars show the range of performance depending on the different data sets. The very left column shows the standalone baseline of both workloads. Running on separate sockets allows the device workload to reach the maximum performance but reduces HLT1 performance to a single socket, which is around 62 - 65% of the full server baseline performance. Running on the full server, including S0.2, favors HLT1, where it marginally misses its baseline performance. However, in that case, INTRP only reaches 36 - 56% of the baseline performance and POLL 59 - 78%. For both POLL and INTRP, the full server without node S0.2 has the fairest performance distribution between HLT1 and device workload. POLL and HLT1 reach a performance relative to their baseline of 74 - 87% and 86 - 88%, respectively, while INTRP and HLT1 reach 52 - 78% and 83 - 85%, respectively. Thus, co-scheduling HLT1 with the compression accelerator reaches the fairest and most efficient performance when using the full server excluding node S0.2 for HLT1 and using polling to communicate with the accelerator.

To evaluate the policy's performance for `mtCPUComp` are the best performing configurations shown in Figure 5.6. This means that the full server performance for `zlib level 2`, `xz level 5` and `zstd level 2` is shown, and additionally the separate socket performance for `zstd level 2`. Same as in the previous figure, the very left column shows the standalone baseline of both workloads. For `zlib level 2` the policy chose to use the full server and interrupts as communication schema. The results show that using INTRP has a minimal advantage over using POLL. For `xz level 5` the policy chose the full server and polling as communication schema. As `mtCPUComp` internally uses NUMA-binding, it is not possible to omit node S0.2 for `mtCPUComp`. Nevertheless, POLL is superior to INTRP. POLL reaches about 60 - 92% of its baseline and 17 - 50% of `xz`'s baseline performance. With POLL is `xz`'s range of performance larger than when co-scheduled with INTRP. The minimum is 3%-points lower than the baseline, while the maximum is up to 15%-points higher. At the same time, the accelerator performance of POLL (60 - 92%) is significantly better than for INTRP (53 - 61%). As the baseline performance of `xz` is so low to begin with, using separate sockets deteriorates the performance of `xz` even more, resulting in a less fairly distributed co-scheduling performance.

The results confirm that for `zlib level 2` and `xz level 5`, the policy results in the optimal co-scheduling performance when wanting to fairly distribute the performance between the workloads.

`Zstd level 2` is a more complex case, as it is both memory-heavy and compute-heavy. The policy recommends distributing `zstd level 2` on the full server, omitting node S0.2. Though it was very close to getting the recommendation to run on separate sockets. For this reason, both performances of running on the full server and running on separate sockets are presented in Figure 5.6. Similar to `xz`, it is not possible to omit S0.2. On the full server, POLL and `zstd` achieve a performance of 72 - 90% and 29 - 60%, respectively. INTRP and `zstd` achieve nearly the inverse performance with 28 - 62% (INTRP) and 69 - 94% (`zstd`), respectively. Co-scheduling POLL and `zstd` has a smaller variance and should be therefore preferred. It means a more robust performance independent of the data input. Using separate sockets achieves the maximum performance for the compression accelerator, with POLL having less variance (minimum 97% vs 95% of the baseline performance). At the same time, `zstd` only achieves 28 - 53% of the baseline. Running on separate sockets is, therefore, not a fairly distributed performance. For the full server co-scheduling configurations, INTRP or POLL, are both valid choices. INTRP favors `zstd level 2`, and POLL favors the device. Using the full server and polling should be the final decision with the argument of performance robustness.

5.7 Discussion

The co-scheduling policy we designed achieves an efficient system performance while also fairly distributing it between host and device workloads. It attains this optimized performance by using NUMA-binding and selecting the superior communication schema for the device. No code changes to the host workload are needed. Additionally, the policy needs only as few as two performance metrics: memory bandwidth and power consumption. These metrics are enough to categorize the compute and memory footprint of a workload.

Using CPU power consumption as a metric to classify the compute intensity is taken based on the problem of acquiring accurate compute-intensity metrics. Compute intensity is a notoriously difficult metric. The modern CPU architectures are complex contraptions that (ab)use registers to optimize the performance. For example, scalar double values might be calculated in vector registers if all scalar registers are in use. While vector registers are often more expensive in calculation time for a single value, it might still be faster than waiting for a scalar register to become available. The calculation of the number of operations executed, and their costs are, therefore, more a *guesstimate* than an accurate measurement. Another option to measure the compute intensity is integrating counters within the program. However, this needs access to the source code and will result in a slight performance degradation due to the additional work to accumulate the counters. For our level of accuracy in classifying the compute intensity, we show that power consumption is accurate enough. Using just the CPU power consumption comes with the advantages of being easily obtainable and being software-independent.

The question might be raised about how, during polling, the latency between requests influences the accelerator's performance. AHA378 has an option in its configuration to throttle the interrupt requests; or in other words, it allows to control the minimum latency between responses. By default, it is set to 200, but the AHA experts suggested changing it to 400 or 600. This indeed results in a higher throughput that is stable. However, it is impossible to fix the latency any other way for polling. This is due to the implementation. To get the maximum performance, many compression streams must be run within the same `compressThread`. Because it is unclear which order they finish, they are looped over in sequential order to find the next compression stream that has finished processing. Introducing sleeps within this loop only resulted in performance degradation.

In Table 5.5, lines 4/5 and 10/11, the co-scheduled MLC performance decreases for the same NUMA-binding when SMT2 is used instead of SMT1. The same also happens to the memory bandwidth. The difference in memory bandwidth can be explained by it being mainly influenced by MLC's performance. The degradation of MLC's performance is likely due to MLC having twice as many threads for SMT2 than for SMT1. MLC can saturate the memory bandwidth with SMT1. Using SMT2 likely results in CPU stalls due to the higher scheduling load of threads requesting memory accesses.

The performance of `HLT1` on separate sockets in Figure 5.5 shows that POLL performs better than INTRP. This is the case, even though `HLT1` is executed only on the other socket, independent of the accelerator workload. POLL most likely performs slightly better than INTRP because interrupts are shared system-wide. As such, interrupts within `HLT1` influence and degrade the performance of INTRP slightly.

For this policy, we have not clearly defined at which percentage of memory bandwidth *oversaturation* is too much, and the workloads must be run on separate sockets. However, the results of `HLT1`, `xz level 1` and, especially `zstd level 2` show that with 119% oversaturation of the single socket, a limit is nearly reached where co-scheduling using the full server becomes inefficient. We expect that the limit is around 120 – 125% when separate sockets become more efficient than using the full server.

The previous section showed that the policy always selects the right combination of NUMA-binding and communication schema to achieve the most efficient, fairly balanced system performance. Figure 5.7 shows the performance increase of applying the policy in comparison to using no policy and no NUMA-binding. For this, `mtCPUComp` is refactored to use no internal NUMA-binding. For simple cases, when host workloads are either memory-heavy or compute-heavy, the policy always increases the performance. The configuration selected by the policy increased the device performance by a factor of 1.5 – 4.0. And for the host workload, it increased by 1.8 – 2.3. The performance increase is larger for the policy configuration than for other configurations.

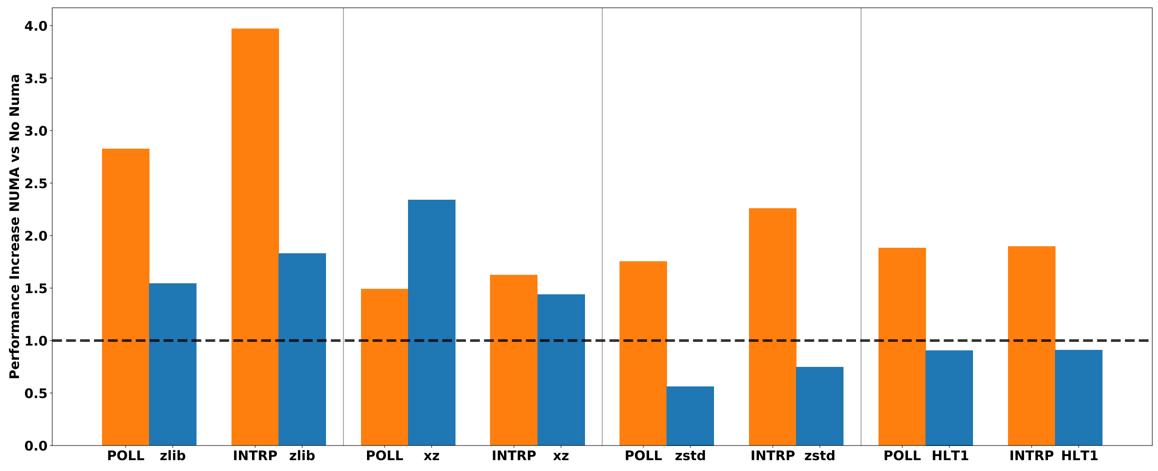


Fig. 5.7 Performance increase using the policy compared to using no policy and no NUMA-binding.

For complex cases, memory- and compute-heavy, the policy has a significant performance increase for the device. While in contrast, the performance of the host workload (`zstd level 2` and HLT1) decreases. The performance increase of the accelerator is between a factor of 1.8 - 1.9 for the selected policy configuration (POLL). At the same time, this configuration has the largest decrease in the host workload performance. The decrease is around a factor of 0.1 for HLT1 and up to a factor of 0.4 for `zstd level 2`. This is not surprising as the policy enforces a fair performance distribution between host and device workloads, even though the host workload already occupies many CPU resources. Therefore, it is unsurprising that the enforced reservation of CPU resources for the device decreases the host workload performance.

Overall, the policy achieves good results with little effort for various workload characteristics. However, the limitations are also clearly visible: host workloads with a high compute intensity, and high memory bandwidth make it challenging to optimize and fairly share the system performance.

While we showed that commercial compression accelerators are a viable solution, they might not always be an option. Sometimes neither of the so far presented solutions are an option: using general-purpose compression algorithms or using commercial compression accelerators. This can be due to minimum requirements of compression ratio or throughput, which the general-purpose compression algorithm cannot please in the required configuration. One further option to solve such limitations is to use compression algorithms that integrate domain knowledge. They can be trained on the specific data to be used on. Choosing the correct compression algorithm for a data set should increase the compression ratio without being so strongly penalized in the throughput as general-purpose compression algorithms are. This will be presented in the next core chapter.

Chapter 6

Compression With Domain-Knowledge

6.1 Introduction

The best compression results are achieved when optimizing compression algorithms for the data they are used on. They take advantage of the underlying patterns within the data by enriching the algorithms with domain knowledge. Depending on the approach, they can be based on index-encoding of the symbol dictionary, use the frequency distribution of the symbols, or transform the data into a different space to condense the information into fewer dimensions. The latter is often lossy compression.

In this chapter, we will explore two different techniques: 1) lossless Huffman encoding with and without delta compression, and 2) lossy ML-based compression technique using autoencoders that transform the data into a compressed, *latent* representation.

Huffman Coding is selected as it allows compressing sequences close to their entropy, maximizing the lossless compression performance. Our expectation of delta compression is that it improves the performance even further by reducing the data range while having a small computational overhead. For the autoencoders, an architecture is selected that behaves similar to the Principal Component Analysis (PCA). PCA is a transformation that allows reducing dimensions by aggregating most of the information in the first few dimensions.

6.2 Related Work

Many general-purpose compression algorithms have their origin in algorithms that are not adaptable but bound to a specific use case. For example, the Lempel-Ziv family, which includes *deflate*, is an adaptive dictionary compression technique. A static version would be digram coding [102], which extends a dictionary by digrams (pairs of symbols) of frequently used combinations to reduce the overall encoding length. In general, domain-specific, non-adaptive compression techniques are the first ones being developed and are more various in possibilities than adaptive approaches. The optimal non-adaptive compression technique will consistently outperform a generic solution in speed and/or compression ratio. However, finding the most performing technique, its proof of implementation correctness, and its maintenance are more challenging than using well-proven general-purpose compression algorithms.

Various algorithms, including partially adaptive ones, benefit from integrating domain knowledge. The most frequently used one is Huffman Coding [102]. It is an entropy coding technique that allows encoding the most frequent symbols by the shortest code words. Due to this, it is often combined with other compression techniques and is included as the last step of many general-purpose compression algorithms, like deflate [30] or zstd [115]. Delta coding, also called delta compression, saves only the delta between two sequentially following elements. It is used, e.g. as a possible compression technique in HTTP [80]. For predictive or context-based encoding, there is, e.g., Prediction by Partial Matching (PPM). PPM performs compression based on the likelihood of having a specific n-long sequence of symbols (n-level context). It is adaptive and allows generating and updating the likelihood distributions. It also has an escape symbol to escape to a lower-level context until down to the alphabet. While PPM is adaptive, its performance benefits if the contexts are already predefined. PPM has multiple applications, including the works of Wu and Teahan [110] who create a new PPM version to compress the Chinese alphabet.

There are also various lossy compression algorithms. Quantizers compress by grouping a data range into one value and reducing the total number of possible values. For example, Cárdenas-Barrera and Lorenzo-Ginori [17] use them in combination with Huffman coding to compress medical electrocardiogram signals. There are also many transformation functions that transform the data into a different space that allows the reduction by having the majority of information in a few elements. They have names like Fourier Transformation, Discrete Cosine Transformation (DCT), Principal Component Analysis (PCA), or Wavelets. Works range here from Zheng et al. [114] that use PCA on distributed, wireless seismographs to reduce the data transfer rates, to Elaskary et al. [35] that use DCT with quantized Huffman or run-length encoding for medical electroencephalography signal compression, to Goldberg et al. [40] that use wavelets to compress digitized radiographs and show that medical professionals did not find any clinically relevant degradation while the compression ratio was below 30.

There are also machine learning (ML) approaches for compression, mainly using autoencoders. Autoencoders try to learn a representation of the input data by ignoring unnecessary information. The layers tend to be mirrored in size; the input layer equals the output layer size, and in the middle are the layers of the smallest size. This smallest, latent representation of the data is then the compressed representation. Most works apply autoencoders to be able to compress images and videos. For example, Zebang and Sei-ichiro [112] use a densely connected autoencoder to achieve superior image compression compared to JPEG and JPEG2000, Theis et al. [108] present an autoencoder that compresses high resolution images at a similar performance as JPEG2000, and Cheng et al. [21] use a complex deep convolutional autoencoder to compress images. Furthermore, Golinski et al. [41] use an autoencoder to perform lossy video compression. Other use cases of autoencoders include the works of Perera and Mo [85] who compress ship performance and navigation data to reduce the data transfer between ship and on-shore data centers. The compression would then allow for increasing the sampling rate. Kaiser and Bengio [56] enhance text translation by first transforming data into a discrete representation to improve the latent space representation of the autoencoder, which in the end increases the overall translation performance. Furthermore, Islam et al. [54] use a deep learning model, including autoencoders, for molecular biology to impute missing values and compress genome expressions. Last, as an example, a recent work of Liu et al. [71] apply a 7-layered autoencoder to different scientific data sets. The data sets consist of double-precision floating-point numbers that can be compressed $2 - 50\times$ better than their conventional counterparts SZ and ZFP while having half the throughput speed.

While there are works on autoencoders that outperform conventional compression algorithms, many works state that either the throughput is lacking or conventional algorithms reach equal performance. Many works also note the importance of pre-processing the data to transform it into a representation to achieve the best performance. A study by Seger [103] shows that for discrete data, One Hot Encoding achieves the best results. However, One Hot Encoding is problematic for large data sets. They tend to explode in size because every value is transformed into a bit mask the length of all possibilities of that value.

As there are too many algorithms available, we select only two to evaluate: 1) a classic lossless, simple approach using delta coding and Huffman coding, and 2) a lossy *en vogue* approach using autoencoders.

6.3 Considered Data Set: VELO Subdetector

The LHCb raw detector data of a single event consists of multiple data packages of different subdetectors concatenated in an arbitrary order. Therefore, exploiting domain knowledge for its compression is best done on subdetector level. For this, we first analyze the subdetectors to find a likely candidate. Afterwards, we present its data format and its data distribution

Table 6.1 LHCb data sets and their most contributing subdetectors

Subdetector	Full File Compression Ratio	Subdetector Part of File [%]	Subdetector Compression Ratio
		LHCb 1	
OT	1.16	26.4	1.06
RICH	1.16	14.4	1.31
VELO	1.16	12.3	1.11
		LHCb 2	
OT	1.14	27.9	1.06
RICH	1.14	14.5	1.32
VELO	1.14	13.5	1.11
		LHCb 3	
OT	1.17	26.2	1.06
RICH	1.17	13.7	1.31
VELO	1.17	12.4	1.11

characteristics. The chosen subdetector should be a significant contributor to be representative but also small in size and easy to understand. This would allow focusing on the compression techniques while the implementation details take a backseat.

6.3.1 LHCb Raw Detector Data

We use multiple LHCb Run 2 data sets from different years to analyze the raw detector data decomposition. Table 6.1 lists the three subdetectors that are the most significant contributors to the file size. All data sets have the same significant contributors: OT, RICH, and VELO. This is expected as the data sets belong to the same detector setup (Run 2), measure the same type of collisions (proton-proton), and contain all subdetectors (*full stream* data).

In this chapter, we will only work with the VELO subdetector. The decision is taken as VELO is the third-largest contributor to the file size, and its compression ratio is below average compared to the entire file. In addition, an expert is available to help understand the details of the data format. OT might have been a more likely candidate, as it is the most significant contributor to the file size and has the lowest *below average* compression ratio of the three subdetectors. However, to evaluate the domain-specific compression performance as a pilot study, OT is unattractive. The data format is more complex, and no expert is available to support the development. RICH is not considered as the compression ratio is above average, and also, there is no expert available.

6.3.2 VELO Raw Data Format

The VELO data format [34] consists of three sections: Raw Bank Header, VELO-specific header, and VELO data. Figure 6.1 shows the layout.

31 … 24		23 … 16		15 … 9		8 … 0		
Total Event Size (includes padding)				Magic Pattern (0xCBCB)				Raw Bank Header
Source ID (DAQ read-out card)				Version (DAQ Software)		Type (Subdetector)		
R(eserved)		PCN		Number of Clusters				VELO Header
Cluster 1				Cluster 0				#Cluster * (Cluster Centroids + sOver)
sOver 1 Bit	Cl. Size 1 Bit	Cluster Position 14 Bits		sOver 1 Bit	Cl. Size 1 Bit	Cluster Position 14 Bits		
Padding				Cluster 2				Cluster Centroids = Cl. Size + Cluster Pos.
				sOver 1 Bit	Cl. Size 1 Bit	Cluster Position 14 Bits		
EOC “0”	ADC0 Cluster 1	EOC “1”	ADC2 Cluster 0	EOC “0”	ADC 1 Cluster 0	EOC “0”	ADC0 Cluster 0	ADC Values
Padding		EOC “1”	ADC 1 Cluster 2	EOC “0”	ADC 0 Cluster 2	EOC “1”	ADC 1 Cluster 1	
31	30 … 24	23	22 … 16	15	14 … 9	8	7 .. 0	

Fig. 6.1 VELO Raw Data Format

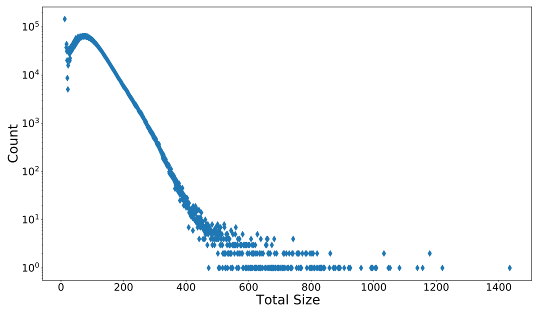
The Raw Bank Header is generic and the same for all subdetectors. It is 8 bytes large and starts with 2 bytes of a magic pattern (0xCBCB) to signal the start of an event. This is followed by the total package size (2 bytes), which includes the size of the Raw Bank Header and the padding at the end of the event. The padding aligns the start of each event at the 32-bit boundary. After this, the (subdetector) bank type and the DAQ software version are described (each 1 byte). The last field of the header is the source ID. It is made up of 2 bytes and identifies the read-out card. Magic pattern, bank type, and version are static for a selected bank type.

The VELO header is 4 bytes large and consists of 2 bytes describing the number of (particle) clusters registered, 1 byte for the Beetle pipeline column number called PCN, and 1 byte of reserved bits. The VELO data follows this. For each cluster, there will be a total of 2 bytes that encode where the cluster centroid is located and if there is spill-over from the previous event or noise. If the number of clusters is uneven, padding of 2 bytes is added at the end of this block of cluster centroids to align it at the 32-bit boundary. The last part of the VELO data consists of the ADC values. They contain the charge measured for a specific cluster. Each cluster can have 1 to 4 ADC values. Each ADC is 1 byte large, with the first

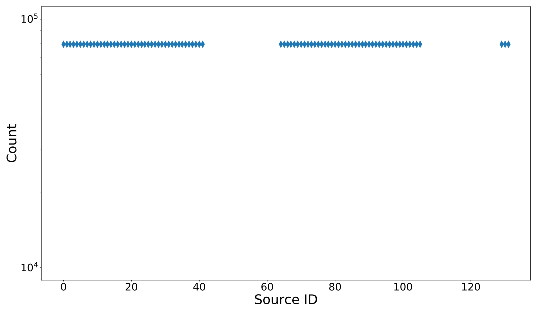
7 bits containing the charge. The MSB (field name "EOC"), if set to "1", indicates that the next ADC value belongs to the next cluster.

6.3.3 Characterization of VELO Data

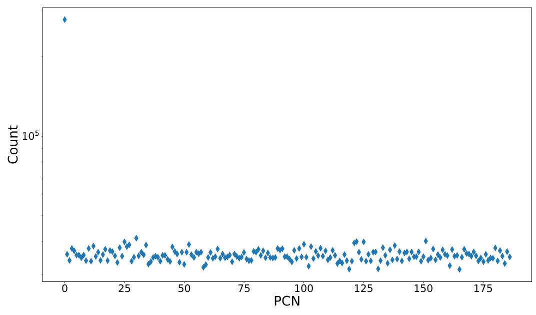
To better grasp what kind of data we are dealing with, we analyze the data distribution of each VELO field. Figure 6.2 shows those distributions for all fields with more than three different values.



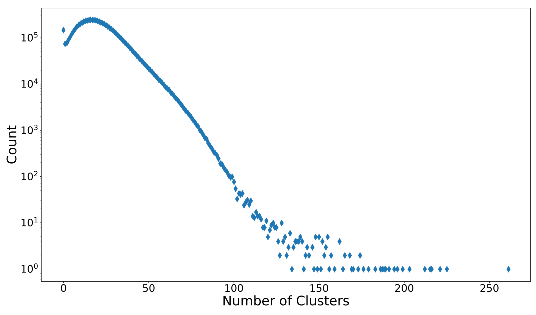
(a) Total Size



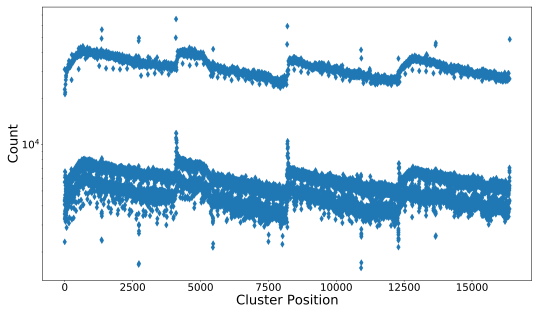
(b) Source ID



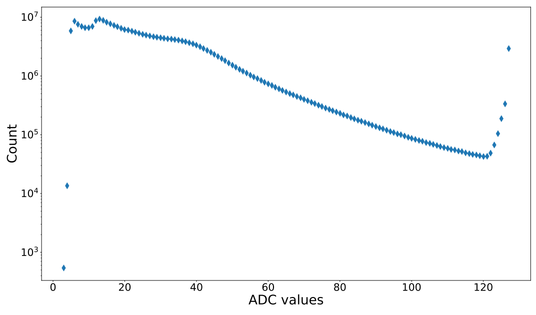
(c) PCN



(d) Number of Clusters



(e) Cluster Centroids



(f) ADC values

Fig. 6.2 Data distribution of VELO fields

Total Size (Subplot a) and *Number of Clusters* (Subplot d) have a similar distribution. It looks like the right-half of a Gamma distribution that fan outs at high x-values. There is also a peak at the first data point: the total event size of 12. These events consist only of the header and have no clusters. Therefore, they contain no data of interest. They make up around 42.9% of all events. The similarity between *Total Size* and *Number of Clusters* makes sense as the number of clusters mainly influences the total size - and with this, the possibility of up to 4 ADC values per cluster. *Source ID* (Subplot b) is limited to fixed values that occur all with the same frequency. This is due to *Source ID* describing which read-out card is being used - and each read-out card delivers data for each event. *PCN* (Subplot c) has one significant outlier with the value "0". The other values are overall homogeneously distributed. The *Cluster Centroids* (Subplot e) are also homogeneously distributed over the value range. However, a pattern in the occurrence of each value is visible. It consists of three bands and four segments. The bands seem continuous, meaning that there exist values that have a high, middle, and low count on regular intervals. Furthermore, the four segments repeat how often a value occurs. Each segment starts with a high count that gets even higher until it dips to fall in frequency of occurrence. The *ADC values* (Subplot f) start with a high counter for low ADC values that slowly falls in occurrence until it reaches its lowest point nearly at the end of the data range. There, the few last values exponentially increase their occurrence.

This characterization of each VELO field shows that they have rather different underlying distributions. For compression, this would mean that either each field needs its own modeled distribution or a more generic approach should be used that does not require such fine-grained tuning.

6.4 Huffman Coding and Delta Compression

This section analyzes the performance of the *classic* lossless compression approach using Huffman Coding and delta compression. The choice of using Huffman Coding is based on its unique properties of being an entropy encoder and its popularity in being used as the last step in many compression algorithms. Delta compression is selected with the expectation of shrinking the domain size, its implementation simplicity, and low compute effort that should result in a fast algorithm that fulfills the throughput requirements of the DAQ.

We first introduce the algorithms, followed by the implementation details and the achieved results from Huffman Coding on delta encoded values and Huffman Coding on original *absolute* values.

6.4.1 Algorithms

Huffman Encoding

Huffman Codes belong to the group of entropy encoders and are optimal prefix codes [102]. The optimum refers to the fact that in a list of symbols that occur in different frequencies, Huffman Codes encode the most frequent symbols in the shortest code words and guarantee that the two least frequent symbols have the same code length. Therefore, the average code length of a Huffman Code is *optimal* and in between the *entropy of the source* and *entropy of the source + 1 bit*. Being a prefix code, it also guarantees that each code word is uniquely identifiable. This means that longer code words never have the same beginning as shorter code words. Thus, the prefix of a code word is never shared with any other code word making each code word uniquely identifiable during decoding. Visualizing this as a tree shows that the symbols are always the leaf nodes and never the internal nodes of the tree.

By definition, a Huffman tree is a binary tree with all the symbols as its leaves. It is constructed as follows: Starting with the two symbols having the least frequency, an internal node is created to combine them. This internal node has then the combined frequency of the two symbols. Afterwards, the two nodes (leaf or internal) with the least frequency are combined into a new internal node. This is done until all leaf nodes (= symbols) are part of the tree.

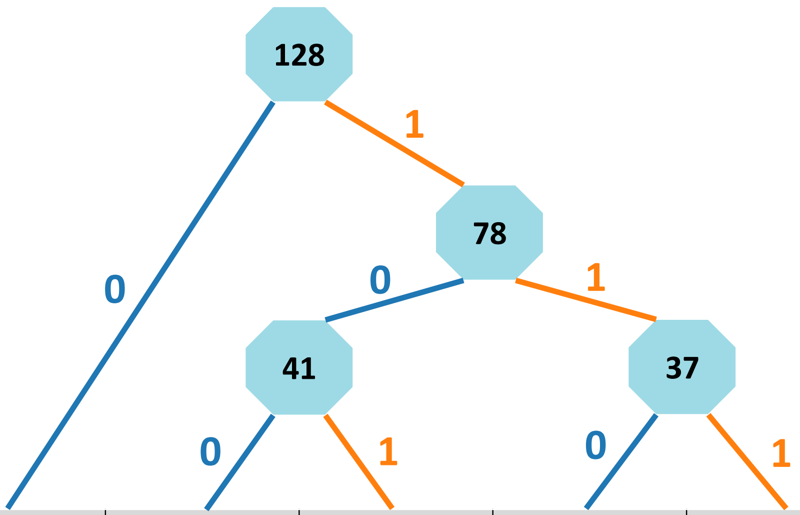
The code word is then identified by traversing the tree from the root until reaching the leaf node with the desired symbol. Selecting the left or right edge concatenates either a "0" or "1" to the code word until the leaf node is reached. An example of this is shown in Figure 6.3.

Delta Compression

The goal of Delta Compression (DC) is a reduction in domain size. This is done by encoding the difference between two data points that follow each other in sequential order. To reconstruct the values correctly, the start value must be saved, and the sequence must be created by subtracting the current value from the previous one.

$$\begin{aligned} &\textit{Encoding} \quad a_\Delta = a_n - a_{n-1} \\ &\textit{Decoding} \quad a_n = a_{n-1} + a_\Delta \end{aligned}$$

In combination with Huffman Encoding, DC does not need to result in an absolute reduction of domain size but needs to at least result in a perceived reduction of domain size.



Symbol	A	B	C	D	E
Occurrences	50	21	20	19	18
Code Word	0	100	101	110	111

Fig. 6.3 Example of a Huffman Tree

This means that the frequency distribution of the DC-domain must result in an on-average reduction in symbols being used compared to the original domain.

6.4.2 Methodology

Hardware

The analysis is performed on a so-called *storage server* which has the specifications listed in Table 6.2. It is the same server as used in the next chapter for the cost-benefit analysis of the LHCb DAQ.

Software

Frequency Table Creating the frequency table is implemented in Python. While looping over the VELO data, a counter is increased for each combination of field name and its delta values. The values of the fields *Cluster Centroid* and *sOver* are combined into one entry in the frequency table with two delta values. The same applies to the fields *ADC* and *EOC*. At this point, the entries are only ordered in alphabetical order by field name. The table is created once and saved as a .csv-file. An excerpt of this is shown in Listing 6.1. As such, the slow but more convenient Python implementation is acceptable.

Table 6.2 Server used for Delta Compression and Huffman Encoding

	Storage Xeon
Architecture	Intel
Platform	x86_64
CPU Type	Gold 6342 @ 2.80GHz
Release Date	2021
TDP [W]	230
Virtual Cores	96
Threads per Real Core	2
Real Cores per Socket	24
Sockets	2
NUMA Nodes	2
RAM [GB]	256
Memory Channels per Socket	8
Max. Memory Bandwidth per Socket[GB/s]	204.8

Listing 6.1 Excerpt of the delta compression frequency table

```
Name,DeltaDistance1,DeltaDistance2,Occurrences
ADCandEoc,-124,-1,1
ADCandEoc,-124,0,3
ADCandEoc,-124,1,3
ADCandEoc,-123,-1,14
...
Pcn,-1,,886
Pcn,0,,6632539
Pcn,1,,903
Pcn,2,,868
...
```

Huffman Tree and Lookup Table The implementation of the Huffman Tree is inspired by the tutorial of Aashish Barnwal [11]. Our implementation has significant changes, though. First, we developed a more object-oriented C++ design by integrating all functions into classes and naming classes more descriptively. Then, we extended the `Node`-class. In the beginning, it only stored the frequency and the *symbol* as an integer. We extended it to take templated tuple data as *symbols*. We further introduced the creation of a lookup table *symbol* $\rightarrow$ *code word* to speed up the encoding process. And lastly, we introduced a `getLeafForCode` function that allows us to decode a sequence of arbitrary many code words, as long as each code word is shorter than 64 bits. It needs the start position of the code word: current index in C-array and bit-offset within that index position. If it successfully finds a code word, it updates the current position and the bit offset based on the number of bits traversed during the assembly. This allows `getLeafForCode` to be immediately invoked with the changed parameters to retrieve the next code word. If the given starting point is not part of a valid code word, it returns a null pointer and sets the bit offset to an invalid value. In that case, the current position is not touched.

DeltaHuff The program `DeltaHuff` consists of two parts: for encoding `DeltaHuffComp` and for decoding `DeltaHuffDecomp`. `DeltaHuffComp` compresses VELO data by first calculating the delta between two events and then using Huffman encoding on that delta. This Huffman sequence is then saved. The Huffman encoding uses the lookup table of each Huffman Tree. For this, the frequency table is loaded once and sorted for each field by ascending usage frequency. Then, the Huffman Tree and its respective lookup table are created based on this. Entries with only one member are excluded. They are: magic pattern, bank type, and version. The first event is saved without being encoded. This is needed as the first delta will be based on it, and the omitted data also needs to be able to be reconstructed. For all remaining events, the delta is calculated for each field based on the previous event, and the Huffman lookup table of that specific field is used to encode it. If the previous event has fewer clusters or ADC values than the current one, their value is replaced with a zero to allow the correct reconstruction. An extra byte is added at the end of the sequence. It contains the bit offset within the last byte of the encoded sequence so that the decoder knows when to finish the decoding process.

The decoding with `DeltaHuffDecomp` works similarly. The frequency table is loaded, sorted, and Huffman trees are created. The last byte is used from the encoded sequence to get the bit offset in the last encoded byte. The first event is copied unmodified to the decoded array. Afterwards, the respective Huffman trees decode the code words to get the delta values.

The absolute value is then calculated based on that delta and the previous (already decoded) event.

6.4.3 Results

Delta Compression with Huffman Coding

Employing delta compression with Huffman encoding (DC&HE) results in a compression ratio of 1.13 for the VELO data. Figure 6.4 shows the performance of DC&HE and general-purpose compression algorithms from the previous chapters. The performance is acquired using the default command-line tools to compress LHCb 1 - VELO. Symbolized as a gray triangle, DC&HE has a weak performance, both in throughput and compression ratio, compared to general-purpose compression algorithms. It compresses similarly well as `zlib level 9` but is 3× slower. It is just a bit faster than `xz level 1` that achieves a better compression ratio of 1.15.

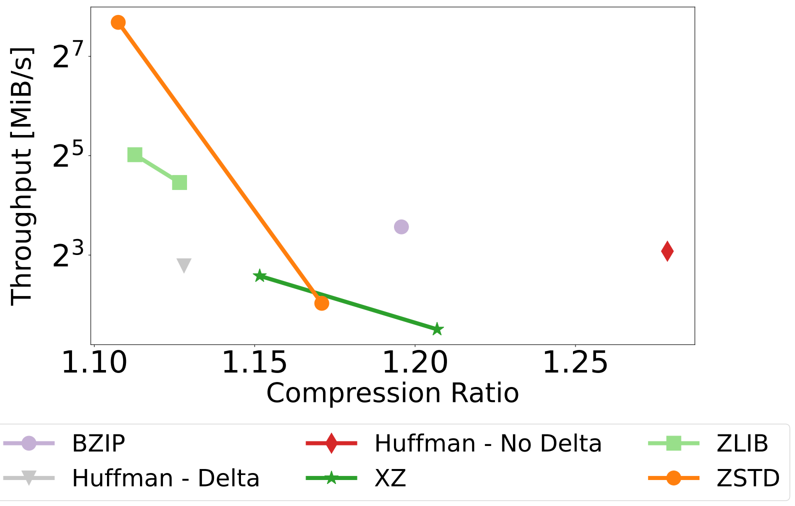


Fig. 6.4 Compression performance of different algorithms on LHCb 1 - VELO

Table 6.3 lists the streaming performance of DC&HE using a modified version of `mtCPUComp` and the streaming performance of general-purpose compression algorithms as used in the previous chapters. DC&HE has a power consumption of 89% of the TDP and a very low memory bandwidth of 3.6 GB/s. Its power consumption ranks in the middle field of the algorithms. For the memory bandwidth, it has the lowest. Even the closest one, `zlib level 2`, has a 3× higher memory bandwidth. DC&HE achieves the best performance when using SMT1 (all real cores). As this performance is not very competitive, we will also evaluate the performance of only using Huffman encoding without delta compression.

Table 6.3 Streaming performance of compression algorithms

Algorithm	Level	#Threads	Throughput [GB/s]	Compression Ratio	Power [W]	Memory Bandwidth [GB/s]
General-Purpose Compression Algorithms						
Bzip2	9	72	0.51	1.20	459	47.4
Lz4	2	48	33.59	1.01	450	128.1
Snappy	-	48	49.65	1.00	380	162.4
Xz	1	96	0.22	1.15	474	61.0
Zlib	2	96	1.93	1.11	471	9.4
Zstd	2	48	8.60	1.11	458	45.5
Huffman Encoding						
Delta	-	48	0.31	1.13	456	3.6
Absolute	-	48	0.35	1.28	456	4.2

Huffman Encoding on Absolute Values

Using Huffman Encoding (HE) directly on the unmodified, original *absolute* values of the VELO data achieves a compression ratio of 1.28. It is the best compression ratio achieved by any of the algorithms tested. Compared to DC&HE, it is 13% faster and compresses 215% better while consuming the same amount of power and a minimally higher memory bandwidth. Compared to the general-purpose compression algorithms, it is between 40 – 260% better in compressing the data. `Xz level 5` is the closest in compression ratio with a performance of around 75% of HE. But, it is nearly 3× slower. The next closest algorithm is `bzip2 level 9`. It achieves about 40% less compression than HE while being 41% faster.

Using HE on absolute values works nearly the same as applying it to delta compressed data. A new frequency table is needed that is based on the absolute values, and also, the `DeltaHuff` needs some modifications. All references to calculating the delta must be removed. The rest stays the same: the first event is still saved unmodified so that in the following events, the magic pattern, bank type, and version can be omitted. Table 6.4 lists a comparison between the number of symbols for the different Huffman trees depending on DC&HE and HE. HE on absolute values reduces the size of the largest Huffman trees (*ClusterCentroidsAndSOver* and *ADCandEoc*) by about $\frac{2}{3}$.

6.5 Autoencoder

This section explores the usage of autoencoders for compressing HEP data. While limited in scope, it will provide us with an idea if or if not using them is worthwhile. We will first introduce the architecture and hyperparameters of the autoencoder, followed by implementa-

Table 6.4 DC&HE vs HE: Unique symbols per Huffman Tree

	#Symbols	
Field Name	DC&HE	HE
Magic Pattern	1	1
Total Size	1187	720
Bank Type	1	1
Version	1	1
Source ID	263	87
#Clusters	293	184
PCN	373	187
Reserved	5	3
ClusterCentroidsAndSOver	156664	45132
ADCandEoc	752	250

tion details of the different autoencoder models and their results. The autoencoder models are chosen to behave similarly to a Principal Component Analysis (PCA), as PCA transformations are well-known for reducing the dimensionality of data with minimal information loss.

6.5.1 Architecture

Foundations

Autoencoder is an unsupervised machine learning technique using neural networks to learn an efficient data representation while reducing information loss. As shown in Figure 6.5, it looks like an hourglass: encoding layers that decrease in size until reaching the tightest bottleneck with the most efficient data representation, followed by decoding layers that increase in size until reaching the original data size. Usually, there are as many decoding layers as encoding layers with the same number of nodes. The *most efficient data representation* tends to be a lossy compression of the data.

There are many ways how to implement an autoencoder exactly. The most appropriate approach for our use case seems to tune the autoencoder architecture to behave similarly to a PCA. The PCA [59] transforms data into a new multi-dimensional space of eigenvectors where the first dimension has the most information, and the higher the dimension, the less information is encoded. Each dimension is orthogonal to the other, meaning they are uncorrelated features that share no information. It is a computational-heavy but powerful transformation that reduces dimensionality with minimal information loss. Interpretation and analysis of the PCA tend to be challenging as fixing the dimensions to concrete model

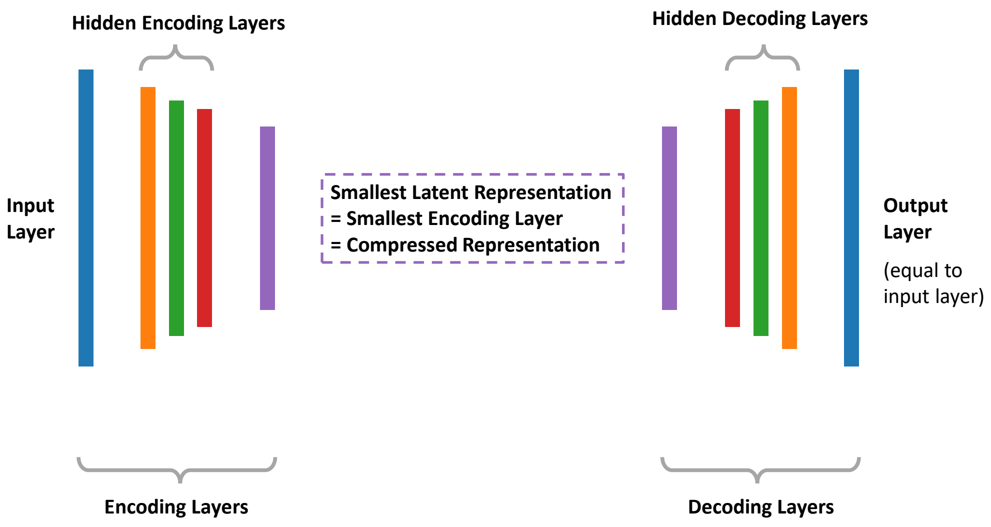


Fig. 6.5 General layout of autoencoders

parameters is, in most cases, not apparent. But this is not of interest for data compression. We are only interested in the smallest representation of data with the least information loss.

Chitta Ranjan [95] provides a tutorial to build such an autoencoder based on PCA principles. He states that a "linearly activated Autoencoder approximates PCA" and that "an Autoencoder extends PCA to a nonlinear space" [96]. For this, the autoencoder uses fully-connected *dense* layers and needs to be enhanced with four features:

- ***Tied weights***: The weights for an encoding layer and decoding layer of the same size are equivalent
- ***Orthogonal weights***: This will make weights simulate to be like the eigenvectors of the PCA
- ***Uncorrelated features***: Each node (here: each field of VELO event) of the compressed representation must be uncorrelated
- ***Unit Norm***: Like the eigenvectors, the weights of each layer must satisfy the unit norm to allow proper estimates

Our autoencoder will consist of three encoding and three decoding layers. The size of the layer will depend on the input size.

Hyperparameters

Hyperparameters are parameters used to set up a specific autoencoder model. For the hyperparameter selection, we let us inspire by popular choices of them. We analyze them using an autoencoder with just a single encoding layer, a sample size of 1.5 million events, run for five epochs, and *mean square error* (MSE) as an evaluation metric. Table 6.5 lists those hyperparameters. Marked in bold are the best-performing hyperparameters that will be used for further exploration of the autoencoder models on VELO data. While the activation function *Linear* is always a bit better than *ReLU*, in some cases, their values are identical up to the 4th decimal position (other activations only up to the 2nd decimal position). Therefore, we also include *ReLU*.

The performance of *Linear* is surprising but goes hand in hand with the statement that PCA-like autoencoders need to be linearly activated. Normally, *Linear* is not a good activator. While it is easy to compute, it has the disadvantage that as soon as a single layer uses it as an activation function, it collapses all layers into a single one that creates a linear dependency between input and output. In general, non-linear activation functions are preferred because they are more flexible and result in better-performing models.

A short introduction of the selected hyperparameters follows.

Activation Functions

1) Linear The activation function is a linear function of the format

$$f(x) = ax + m$$

While simple, it has disadvantages. For example, it is not possible to define it for a specific range, and if any single layer in a neural network uses it as an activation function, the output layer becomes a linear function of the input layer.

2) ReLU Rectified Linear activation function [84] is one of the most common activation functions. It is defined by

$$f(x) = max(0.0, x)$$

meaning that if x is negative it returns zero, otherwise x.

Loss Functions

1) MSE Mean Square Error between predicted $\hat{y}_i$ and original value y_i.

$$MSE = \frac{1}{n}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

MSE is a classical metric to evaluate the accuracy of a prediction. In case it is normalized to be in range [0.0, 1.0] then 0.0 is the best value representing the smallest error and 1.0 is the worst value representing the largest error.

2) Huber Loss [49] Similar to MSE but less sensitive to outliers. The formula [48] to calculate it is

$$L(y,\hat{y}) = \begin{cases} (y-\hat{y})^2 & \text{for } |y-\hat{y}| \le \delta, \\ 2\delta|y-\hat{y}| - \delta^2 & \text{otherwise.} \end{cases}$$

with δ being an additional hyperparameter that needs to be trained.

3) Log Cosh Logarithm of the hyperbolic cosine of the prediction error. It is similar to Huber loss with being resistant to outliers.

$$Logcosh(x) = \log(\frac{1}{2}(e^{-x} + e^{x}))$$

$$or$$

$$Logcosh = \frac{1}{N}\sum_{i=1}^{N}\log(\cosh(\hat{y}_i - y_i))$$

Figure 6.6 shows the general behavior of the loss functions with MSE in red, Log-Cosh in green, and for Huber loss, there are two variants for different δ, one shown in blue and the other in magenta. Additionally, in black is also the mean of absolute error $MAE = |x|$ shown.

Optimizers

1) Adam [58] is a stochastic gradient descent based on an adaptive estimation of the first and second order moments of the gradients.

2) Nadam [33] is the Adam optimizer extended to include Nesterov momentum.

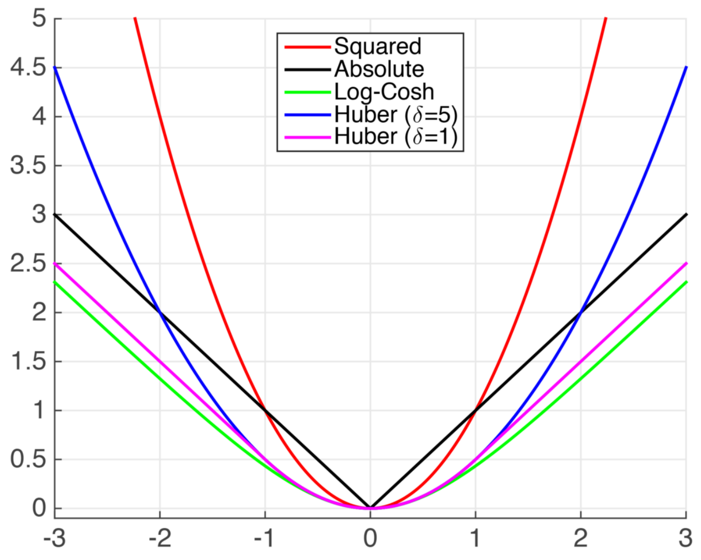


Fig. 6.6 Loss functions: MSE in red, Log-Cosh in green, and Huber loss in blue and magenta [16]. The x-Axis corresponds to the error between the original and predicted value, and the y-Axis represents the loss function for this error value.

Table 6.5 Hyperparameters of the Autoencoder. Marked in bold are the best-performing ones selected for further exploration.

Hyperparameter	Functions
Activation	**ReLU**, **Linear**, sigmoid, softmax
Optimizer	**Adam**, **Nadam**, Adamax, Ftrl, Nadam, RMSprop, sgd
Loss Function	binary cross-entropy, cosine similarity, **Huber**, **Log Cosh**, **mean square error (MSE)**, mean squared logarithmic error (MSLE)

Data Pre-Processing

For autoencoders, like for any classical neural network, the layer size is fixed and, therefore, the dimensions of the input data must also be fixed. This means we need to preprocess the VELO data. VELO events will be grouped by their event sizes and the number of clusters. Then, they will be transformed to be all of equal size for that particular group. Having multiple groups means we have to train multiple autoencoders. We use two different data representations: padded representation and One Hot Encoding.

Padded representation (SLE) This representation will use the original data and introduce padding to organize the input and equalize them to have the same length. Therefore, it will be called *Same Length Events* (SLE). Two sections of padding will be used. One section of padding will be after the cluster centroids. Padding is inserted until the maximum number of clusters for this group is reached. This will allow all ADC values to start at the same position for the autoencoder. The other padding section will be at the end of the event. It allows all events to have the same space for the ADC values as the event with the maximum ADC values.

One Hot Encoding (OHE) As shown by Seger [103], this data representation achieves the highest accuracy for neural networks applied to discrete data. For each input feature (here: VELO field), a bit mask is created in the size of the domain with all bits being zero, but the one hot bit (being "1") representing the original value. For example, if an input feature ranges from 12 to 24 in value, the value 13 would be in OHE a 12-bit long bit mask with the second bit set to "1". The same padding method is applied as used for SLE. While OHE creates good results, it has also disadvantages. It creates complications for input features with a large domain because the resulting bit masks can quickly become too large, creating memory issues and making neural networks large and slow.

Evaluation Parameters

The evaluation uses three metrics: MSE, Accuracy Score, and PSNR. MSE is the same as used for the loss function.

Accuracy Score Metric used for evaluating classification models. The accuracy score describes how many predicted values (classes) are equal to the original value (class). It is a binary evaluation, as the predicted value is either equal to the original class or not. The value is given as a percentage from 0 to 100%.

PSNR [102] describes how much noise is introduced by a reconstruction compared to the original data **Y**. It is a fuzzy metric, as it just has a general notion that the higher the PSNR value is, the better the reconstruction is. It is defined as

$$PSNR = 20\, log_{10}(MAX_{\mathbf{Y}}) - 10\, log_{10}(MSE)$$

6.5.2 Methodology

Hardware

The server used to train the autoencoders is different from the previous section. It is an *EB server* of the LHCb DAQ populated with a single GPU. This GPU is an NVIDIA Quadro RTX 6000 with 24 GB RAM and a TDP of 260 W. This GPU was first released in

Table 6.6 Server for the Autoencoders

	EPYC
Architecture	AMD
Platform	x86_64
CPU Type	EPYC 7502 @ 2.5 GHz
Release Date	2019
TDP [W]	180
Virtual Cores	64
Threads per Real Core	1 (deactivated)
Real Cores per Socket	32
Sockets	2
NUMA Nodes	2
RAM [GB]	512
Memory Type	DDR4
Memory Channels per Socket	8
Max. Memory Bandwidth per Socket[GB/s]	204.8
PCIe	Gen4

2018. Simultaneous multithreading is disabled for this server. Table 6.6 lists all technical specifications of the server.

Software

Data Pre-Processing For each data representation, one Python program is created. For SLE, the first step is to select events based on the eligible criteria. They are minimum and maximum event size and a maximum number of clusters. This selection is based on the characterization of the VELO data. For all the selected events, the ADC values are counted. The new total size of these SLE data sets is adjusted to fit the maximum number of clusters and the maximum number of ADC values. This event size might be larger than the original size, as the events of maximum size do not necessarily have the maximum number of ADC values. The maximum number of clusters is selected to include the large majority of events within the range of selected event sizes. Table 6.7 lists the grouping of the events and their resulting SLE size. Unless the event size is super small or super large, the newly created events are 17 – 28% larger than the original ones.

For the OHE transformation, the first step is to evaluate the domain size of each field for its respective group. Each field within an event is transformed into its bit mask to create the one hot encoded event. The same padding style as for SLE is used: it is added after the cluster centroids and at the end of the file to fit all ADC values. As the OHE event size

Table 6.7 Grouping of VELO events and their SLE size

	Event size			
#	Minimum	Maximum	#Clusters	SLE size
SLE 1	0	60	8	60
SLE 2	61	110	32	139
SLE 3	111	200	60	256
SLE 4	201	400	100	468
SLE 5	401	1434	140	800

Table 6.8 Grouping of VELO events and their OHE size

	Event size			
#	Minimum	Maximum	#Clusters	OHE size
OHE 1	0	60	8	135317
OHE 2	61	110	32	532035
OHE 3	111	200	60	998062
OHE 4	201	400	100	1669559
OHE 5	401	1434	140	2354141

increases by a multiple, a limitation on the total OHE file size is added. It is set to 150 GB. Listed in Table 6.8 are the resulting OHE events.

Autoencoder The first, single-layer implementation of the PCA-like autoencoder in Python is inspired by the tutorial of Chitta Ranjan [95] and uses `TensorFlow` [107] and `Keras` [57]. The architecture is then modified to consist of three encoding and three decoding layers that can be trained on multiple different optimizers, activation functions, and loss functions. GPU support is also added. After running into memory problems, a data generator is added that allows extracting *batch-size*-many sequential events starting at a random position directly from the file without having to load the entire file into memory. At the same time, we extend our program to have a *continuous load* loop that, after every processed batch, loads a new batch directly from the file and, in addition, also checks if the time threshold is passed. If yes, the fitting is stopped, and the model is evaluated. The MSE and Accuracy Score evaluation is done on the entire VELO event and its different sections: *Header* (Raw Bank + VELO), *Cluster Centroids*, and *ADC values*. For OHE, this means that the domain size of each field needs to be known to know where each section starts and ends. Functionality to save the model exists.

Listing 6.2 Software versions used for the autoencoders

```
Python 3.6.8
CUDA version 11.3
Keras Nightly 2.5.0dev
TensorFlow 2.5.0
```

For SLE, the data is first scaled to be in the range of 0.0 – 1.0 (based on all events of that group). The evaluation is done using the *unscaled* data. This means that the inverse scaling function is applied to the predicted data before evaluating it.

For OHE layer sizes, we take a multiple of 2 to reduce the layer sizes to match the requested compression ratio. The first reduction is the largest multiple of 2 possible when splitting the compression ratio over three layers. For the compression ratio 16, each encoding layer size is created by the previous layer size divided by the factors 4, 2, and 2 (in order, starting with the input layer). For the compression ratio 32, the factors are 8, 2, and 2. The SLE layer sizes are more arbitrary. They were custom-tuned to have the smallest layer size be a multiple of 8, 16, 32, or 64 (depending on event size), with the largest possible value not surpassing the average event size of the original event group. The idea behind it is that this compression size results in *real* compression and that the multiple of 8 supports performance optimization of the hardware.

Implementation Notes The tutorial of Chitta Ranjan [95] is written for TensorFlow version 1. However, the server and CUDA version makes it impossible to use TensorFlow version 1. Instead, TensorFlow version 2 is used in compatibility mode for version 1. Limitations of this compatibility mode include, e.g., that the `fit` function does not support data generators and that for saving the model `save_weights` must be used (instead of `save_model`). Listing 6.2 lists the software versions used.

For OHE, the domain size of each field must be known. In theory, it is wise to save this in a file and just load it. For this prototype, we skip this step and recalculate the field size each time exactly once at the program start.

6.5.3 Results

SLE

The padded data set SLE 3 is used to evaluate and reduce the different hyperparameters as stated in section 6.5.1. The selected hyperparameters are then trained on all the SLE data sets based on LHCb 1 and evaluated on the SLE data sets based on LHCb 1, LHCb 2, and

Table 6.9 Best SLE hyperparameters based on MSE performance. *Opt.* stands for optimizer, *Loss* for loss function, and *Activ.* for activation function.

Data	Header			Cluster Centroids			ADC values		
Set	Opt.	Loss	Activ.	Opt.	Loss	Activ.	Opt.	Loss	Activ.
SLE 1	Adam	Huber	ReLU	Adam	MSE	ReLU	Nadam	MSE	ReLU
SLE 2	Adam	Log Cosh	Linear	Nadam	Huber	Linear	Nadam	MSE	ReLU
SLE 3	Adam	MSE	Linear	Nadam	MSE	Linear	Adam	Huber	Linear
SLE 4	Adam	MSE	ReLU	Adam	MSE	ReLU	Adam	MSE	ReLU
SLE 5	Nadam	Huber	ReLU	Adam	MSE	ReLU	Adam	MSE	ReLU

LHCb 3. The standard deviation of the MSE for the different data sets is between 0.01 – 0.51% of the MSE value of the *unscaled* data. The percentage of the difference between the standard deviations increases minimally for larger event sizes compared to the smallest one, SLE 1, (max 10%) even though the event sizes themselves increase by a multiple.

Table 6.9 lists the best performing hyperparameters for each VELO section. The data is acquired by using an older autoencoder implementation (without the data generator). It evaluated only the MSE and used all events of an SLE data set for ten epochs. The total run time for all five data sets is over ten days. Independent of the section, the optimal activation function for the smallest and largest SLE event size is ReLU, while for the medium SLE event sizes, it is Linear. The choice of activation function has the largest influence on the section *Header*, independent of if ReLU or Linear results in the better performance. The factor of improvement between those two activation functions is 2.07 – 9.29 for the better performing one. For section *Cluster Centroids* the factors are 1.12 – 1.57 and for section *ADC values* they are 1.10 – 1.46.

Furthermore, for section *Header* Adam is in all cases, but SLE 5, the best optimizer. For the loss function, there is no such preference. For both sections, *Cluster Centroids* and *ADC values*, there is no preferred optimizer, but MSE is the preferred loss function. Overall, in $\frac{2}{3}$ of all cases is Adam the preferred optimizer, MSE the preferred loss function and ReLU the preferred activation function.

Table 6.10 lists the compression ratio achieved and Figure 6.7 shows the signal-to-noise ratio (PSNR) for the best configurations. For the PSNR, we simplified the calculation of MAX_Y by taking the number of bits of a field to calculate the maximum value instead of its actual range. However, this simplification should only mainly influence the *Header* which is identical in size for each SLE data set. It should change nothing for the relative comparison between the data sets.

There seems to be no relation between compression ratio and PSNR. However, the PSNR gets worse for *Header* when the event size increases, while the reverse is true for *ADC values*.

Table 6.10 Compression ratio for the best SLE configurations

	Data Set				
	SLE 1	SLE 2	SLE 3	SLE 4	SLE 5
Compression Ratio	1.9	2.2	2.9	2.25	2.5

Cluster Centroids has the worst prediction for SLE 1, improves for SLE 2, and plateaus at the maximum for SLE 3, 4, and 5.

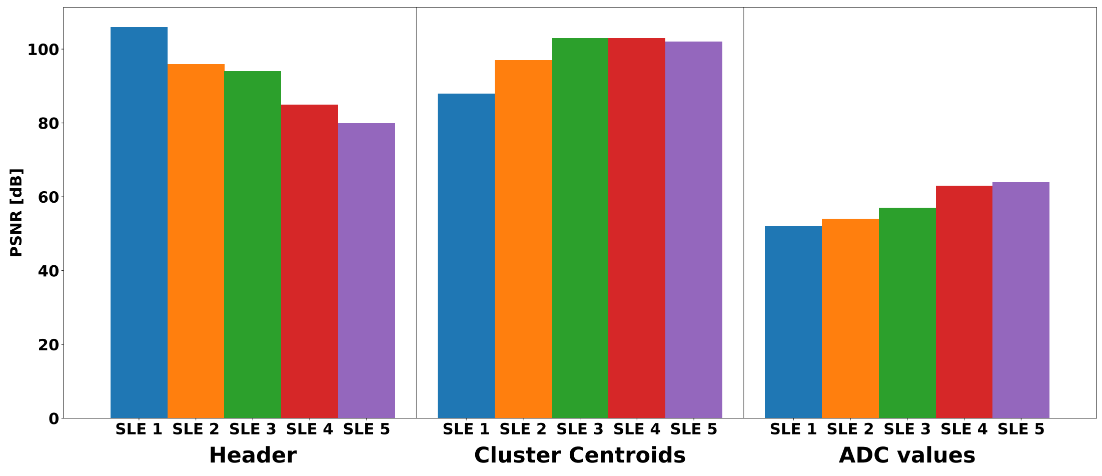


Fig. 6.7 PSNR for the different sections of the SLE data sets

At a later point, after analyzing the OHE autoencoder, an additional run for SLE 1 is performed to evaluate the accuracy score. This time it uses the data generator and is run for 12h on the GPU. Selected hyperparameters are `ReLU`, `MSE` and `Adam`, as they previously performed the best. Because of the development process within the autoencoder, the breakdown of the different layers is not identical. However, we ensured that the smallest layer had the identical size of 32 elements. The evaluation shows that the `MSE` is in equal range for old and newly trained model for *Header* and *Cluster Centroids*. But for *ADC values*, the `MSE` is nearly twice as large for the new model compared to the old model. The accuracy for the new model is for the *Header* 61.5%, for *Cluster Centroids* 41.8% and for *ADC values* 31.0%.

The total number of parameters of the SLE 1 autoencoder is 11,477 of which 51.1% (5,867) are trainable. And the total number of parameters of the SLE 5 autoencoder is 2,020,664 of which 50.1% (1,012,056) are trainable.

Table 6.11 Best hyperparameters for OHE 1

VELO Section	Activation Func	Loss Func	Optimizer
Compression Ratio 16			
Header	ReLU	Huber	Nadam
Cluster Centroids	ReLU	Log Cosh	Nadam
ADC values	Linear	Log Cosh	Nadam
Compression Ratio 32			
Header	ReLU	Huber	Nadam
Cluster Centroids	ReLU	Log Cosh	Nadam
ADC values	ReLU	Huber	Nadam

OHE

The three-layered OHE autoencoder model is run in two configurations of the layer size, with the smallest layer size being 4228 or 8457. This corresponds to a compression ratio of 32 and 16, respectively. For OHE 1 and encoding layer size of 4228, the total number of parameters is 4,935,289,551 with half of them being trainable. For OHE 1 and encoding layer size of 8457, the total number of parameters is 10,585,873,654 with also half of them being trainable. Those large numbers of parameters immediately create issues. The OHE event size is too large to fit the model on a single GPU. Even for the smallest event size, OHE 1, it uses more than 64 GB to save just the weights on the disk. The GPU has only 24 GB RAM. In this case, the GPU segfaults. The OHE autoencoder can only be run using CPUs. However, this also runs into problems. The 500 GB RAM has only enough capacity for the model of OHE 1. Any larger OHE event size is not possible and crashes the program.

Therefore, the OHE autoencoder is only trained using the CPUs and only OHE 1 as the data set. Each hyperparameter configuration is trained for 12h. On average, in those 12 h, only 9766 events (611 batches of 16 events) could be processed for the training. This is less than 1% of all OHE 1 events.

The hyperparameters trained on are the same as listed in Table 6.5, but without the optimizer `Adam`. Using `Linear` as an activation function only works for the smaller compression ratio (layer size 8457). For the higher compression ratio, the `MSE` is as high as $5 * 10^{12}$. For the activation function `ReLU`, both compression ratios achieve high accuracy at first glance. However, as the OHE data inflates the event size by a multiple, it needs a more detailed evaluation. For this, we calculate the average number of bit-flips based on the accuracy score and compare them with the maximum elements (= bits that are "1") per section.

Table 6.11 lists the configuration of hyperparameters with the best results, and Figure 6.8 shows their average bit-flips for the different VELO sections of OHE 1. The number of maximum elements per section is also shown in green. For OHE 1, the *Header* has 8 elements. The *Cluster Centroids* has 16 elements (8 centroid positions and sOver values), and the *ADC values* has 64 elements (32 ADC values and EOC values). The OHE autoencoder compresses the *Header* lossless for both compression ratios. *Cluster Centroids* has the largest error. For both compression ratios, the average bit-flips are more than twice the elements that exist in the first place. Thus, none of them can be accurately reconstructed. On the other hand, the *ADC values* have only up to 2% reconstruction error for compression ratio 16 and up to 4.4% reconstruction error for compression ratio 32.

The difference in compression ratio makes no difference for the *Header*. The smaller compression ratio is 8% better for *Cluster Centroids* and 220% better for *ADC values*.

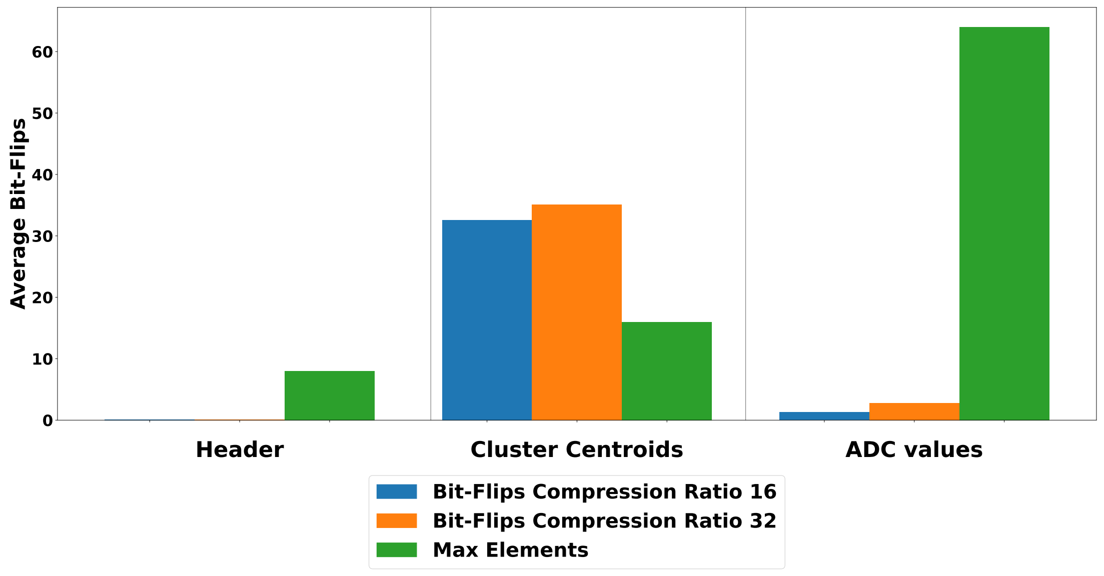


Fig. 6.8 Average bit-flips for the different compression ratios of data set OHE 1. The *Max Elements* in green are the reference of how good or bad the average bit-flip count is. The bit-flip count is based on the accuracy score.

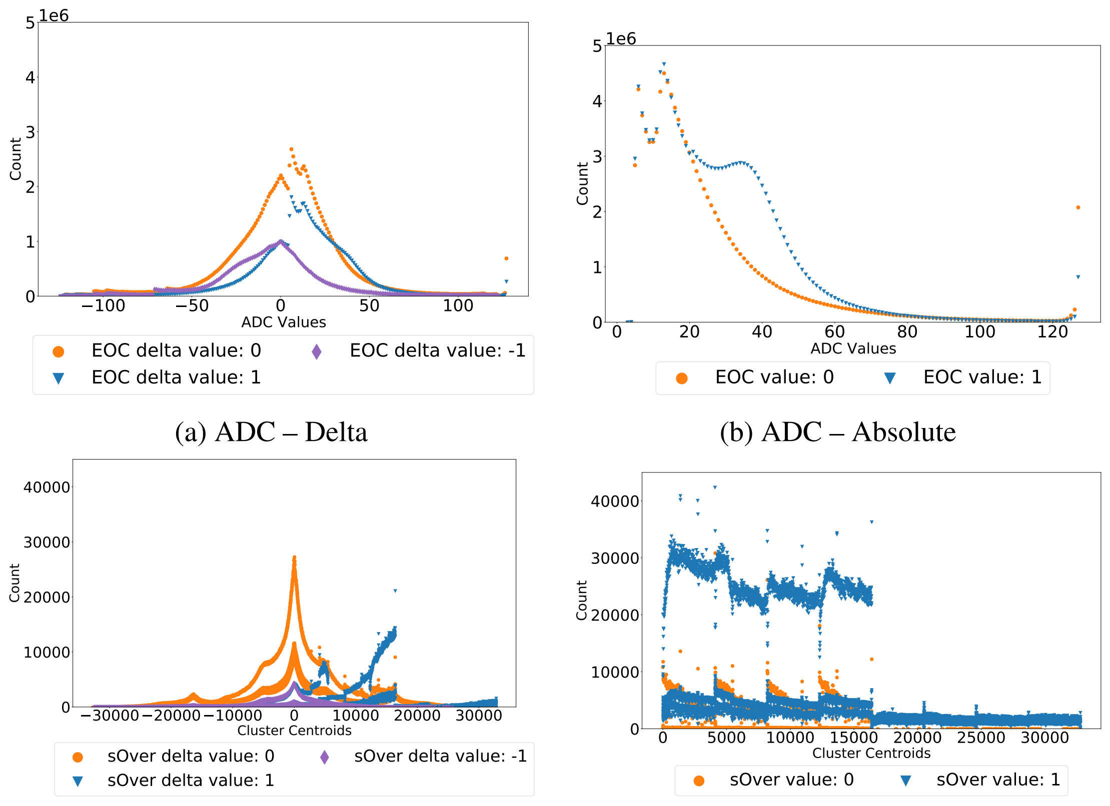
6.6 Discussion

This chapter applied two different compression techniques that use domain knowledge to the VELO raw detector data. In the first part, we analyzed the performance of Huffman encoding on delta encoded values and on absolute values. And in the second part, we analyzed the performance of PCA-like autoencoders using either data only modified by padding to have the same length (SLE) or data modified to be one hot encoded (OHE).

Huffman Coding and Delta Compression

Against our expectations, Huffman encoding on absolute values outperforms the compression performances of all the tested algorithms by 40 – 260% (see Figure 6.4). This includes the combination of delta encoding with Huffman encoding, which is outperformed by 215% in compression performance and 13% in throughput. This is unexpected, as delta encoding tends to reduce the domain size by concentrating the value range around *few* delta values. However, this is not the case here and is most likely a result of the randomness of quantum mechanics measured in HEP. Interactions in quantum mechanics are genuinely random events, and as compression works by pattern recognition, true randomness interferes significantly with its performance. The randomness also results in a large variety of delta values, so much that the number of symbols for the two largest Huffman trees increases by a factor of about three. While this could be, in theory, no problem if only a few symbols have a high usage frequency, it becomes a problem if this is not the case. A big Huffman tree will result in long code words, and as the frequency does not reduce enough, too many long code words are used, and the compression performance is compromised. Figure 6.9 shows the usage frequency for ADC values and Cluster Centroids when using Huffman encoding on absolute values and delta encoded values. Using delta encoding creates a distribution similar to the Laplace distribution with a sharp peak at zero. However, the overall usage frequency of all the values is lower for delta encoded values than for absolute values. This means that the usage frequency of delta encoded values is more homogeneously distributed than the one for absolute values, decreasing the performance of Huffman Coding. A possible solution to improve the performance of delta encoding could be using a different type of Huffman tree. Minimum Variance Huffman Trees [102] are constructed in a way that the variance between code words is minimal while having the same entropy as the classic Huffman tree. This could help with more homogeneous probability distributions.

When it comes to the question of which fields are included in the encoding or not, the fields *Magic Pattern*, *Bank Type* and *Version* are omitted. This is possible as they only have a single entry in the frequency table. On the other hand, the field *Reserved* is included, as it has more than one entry in the frequency table, and while the documentation says it is not used, it might be used at a later point. Furthermore, the decoded events were not identical to the original ones. This is traced back to the type of padding used between the cluster centroids and ADC values. Instead of using 0x0000, it is padded with 0xCCEE. Changing the decoding to use this padding style solved the issue.



(a) ADC – Delta (b) ADC – Absolute

(c) Cluster Centroids – Delta (d) Cluster Centroids – Absolute

Fig. 6.9 Frequency distribution of ADC and Cluster Centroids used by the Huffman coding to either encode based on delta values or absolute values.

Autoencoders

We use two different data representations for the autoencoder, SLE and OHE, and a variation of hyperparameters. Both encodings have advantages and disadvantages. While this exploration struggled with technical implementation details, PCA-like autoencoders still show promising compression results.

SLE allows the training of small models that are robust. The performance differences between various VELO Run2 data sets are so minimal that they can be neglected. While not having the chance to test it, OHE can be expected to create similar robust autoencoders as SLE.

All autoencoder configurations result in lossy compression, as none of them is able to reconstruct events flawlessly. For SLE, we trained two different compression ratios but only presented the better-performing one in section 6.5.3 Results. Contrary to our expectations, the larger encoding layer (worse compression ratio) does not automatically result in better reconstruction performance. SLE3 and SLE 5 achieve a better reconstruction with the smaller encoding layer than with the larger one. Why this is the case is not clear. There was no apparent connection between the encoding layer size and its reconstruction performance. For SLE, using the metric PSNR shows that the *Cluster Centroids* are better reconstructed than the rest. And the *ADC values* have the most noise in the reconstruction. The opposite is true for the OHE autoencoder. The OHE autoencoder can reconstruct the *Header* section without any loss. *ADC values* can be accurately reconstructed between 95.6 – 98%, depending on the compression ratio. But the *Cluster Centroids* are impossible to reconstruct for OHE. The difference in reconstruction performance might be due to the different sizes of the sections. For SLE, ADC values are the largest contributor (45 – 64%) to the event size. For OHE, the header and ADC values make up $< 3.3\%$. The remaining 96.7% of the event are the *Cluster Centroids*. Furthermore, *Cluster Centroids* have a larger value range and are more homogeneously distributed than *ADC values* (see Figure 6.2) which could complicate learning the correct representation. Like with the delta encoding, the bad performance could result from the unpredictability of the quantum mechanic's randomness. In addition, as the OHE 1 autoencoder is only trained on 1% of the data, training on more data will most likely improve the accuracy for *Cluster Centroids*.

Splitting padding into two sections is motivated by the idea that when the *ADC values* always start at the same position, the autoencoder does not associate the same input feature with both *Cluster Centroids* and *ADC values*. *Cluster Centroids* are 16 bits large, and *ADC values* are 8 bits large resulting in greatly different domain sizes. Having them start at different, fixed positions and with this, the association of input features with the same type of data could allow autoencoders to perform better.

Log Cosh, unlike Huber loss, is a double differentiable loss function. This feature can be necessary for certain models, but it also increases the computational complexity. In our case, Log Cosh only outperformed the other loss functions in a single case for the SLE 2 *Header* section. As such, Log Cosh can be deemed unnecessary for this model and can be omitted, reducing the overall computational effort of the model.

The compression ratios we use for the OHE autoencoder are 16 and 32. For OHE 1, this corresponds to a compressed event size of 8457 and 4228, respectively. This is a factor 141 and 70 larger than the original SLE event size for the same group of events. To improve this, additional compression techniques would need to be deployed. Contrary to the original data that are integers, the compressed representation of the autoencoders is composed of floating-point numbers. As such, compression techniques could be employed that work on floating-point numbers, like, e.g., quantizer. However, those techniques are likely to introduce additional losses.

A limitation of our exploration of autoencoders is that we did not analyze the throughput performance. Evidently, smaller models are faster than larger models. But there are no numbers available to compare the autoencoder performance to other compression algorithms to rank the throughput. When grouping by event size, the data transformation is applied directly before being saved to different files. In addition, the SLE performance of the current grouping of events would be unrealistic. SLE 1 needs to be split into more groups to both allow a more realistic compression ratio and throughput. This is due to the fact that nearly half of all events contain no clusters and have a total size of 12 (see section 6.3.3). With SLE 1 being the smallest group, they are all inflated by a factor of 5 to an event size of 60. Just SLE 1 alone contains already 98.1% of all events. An event size of up to 156 covers 99.99997% of all events. Thus, to get an efficient performance, the grouping of the event sizes must be done to keep the inflation of event size to a minimum. This was not done here, and we did not create more groups, as we wanted to see the general notion in the performance of the autoencoder.

There is a further limitation for the OHE autoencoder: its memory requirements. By definition of using OHE data, the network sizes are larger, and with this, the autoencoder is slower than SLE. In our case, the network size reaches such a size that it makes the usage of a single GPU as accelerator impossible. To use GPUs, a distributed autoencoder model would be needed that can split the model onto multiple GPUs. Using just CPUs, the OHE autoencoder is so slow that it did not even manage to train on 1% of the OHE 1 data in 12h, while SLE 1, even on the CPU, had multiple iterations over all its events during a three-hour training time. It also seems that the `Nadam` optimizer uses less memory than the `Adam` optimizer. During testing different layer sizes, it was possible to run larger encoding layers with `Nadam`, but they crashed with `Adam`.

Comparison with Other Neural Networks It is difficult to compare the performance of our autoencoders to any of the previously mentioned works. Compression autoencoders are mainly deployed to compress images, videos, or scientific data based on floating-point numbers. Despite a thorough search, no works were found that apply compression autoencoders to scientific data based on integers, as we did in this chapter. In addition, not all works provide parameter sizes for their autoencoder models, making it difficult to compare with them. For compression autoencoders applied to images, a tendency can be seen in the choice of parameters. Convolution layers are favored over fully-connected *dense* layers, and even if they work with high-resolution images, the autoencoder itself only operates on partial images no larger than 128×128 pixels. This corresponds to a domain size of the input layer of 16,384 or 49,152 for RGB-colored images. Famous image corpora like CIFAR-10 [61] or

MNIST [65] are even smaller. They, respectively, consists of 32 $\times$ 32 color images (input size 3072), or of 28 $\times$ 28 grayscale images (input size 784). For OHE 1, the input layer size is at least 2.75$\times$ larger (135,317). In addition to the smaller layer size, compression autoencoders do not use many layers. As such, the total number of parameters of the other works will be a multiple smaller than for our autoencoders.

Neural networks that have a high number of parameters can, for example, be found for natural language processing (NLP). Famous NLP models are GPT-2 [93] with 1.5 Billion parameters or GTP-3 [15] with 175 Billion parameters. GTP-2 has around 75% of OHE 1-CR32[1] trainable parameters, while GTP-3 is about 35 $\times$ larger compared to OHE 1-CR16[2] trainable parameters. However, in all those cases, the training occurs on supercomputers. For example, GPT-3 had access to a supercomputer with 10,000 NVIDIA V100 GPUs. While no official numbers are available, Narayanan et al. [81] estimate that GPT-3 would need to be trained for around 34 days on 3072 of the better-performing NVIDIA A100 GPUs. Only a few dedicated working groups have access to such kind of hardware resources and compute times. Furthermore, compared to their parameter size, the domain size of their input layer is tiny. For example, GPT-2 only uses 1024 tokens (= words) as input, while another NLP model, T5 [94] that has up to 11 billion parameters, has a maximal domain size for the input of around 65,000 - or about half of OHE 1's input domain size.

Evidently, even if models have the same or more parameters than our models, their layer sizes are significantly smaller. They reach their large numbers of parameters by utilizing many layers (GPT-2 has 118 layers). Therefore, these models spend more time computing while having smaller memory requirements for data processing and layer sizes than our OHE model. This simplifies the model's computation in a distributed, heterogeneous environment. Compression autoencoders for images have the advantage of subdividing images to allow the usage of smaller networks. This is possible because the vast majority of pixels in images have only a small local spatial dependency. For HEP data, this is different. The main dependency is not within a single subdetector but between consecutive subdetectors due to the particles flying from the point of collision through the subdetectors until being absorbed. This means little to no spatial dependencies within a subdetector during a single event. This independence makes predictions autoencoders more difficult as patterns that could be utilized for compression are missing.

After comparing the model architectures, we explore the inference performance of the different models. Comparing with other compression autoencoders is challenging as many do not use accuracy scores as a performance metric but fuzzier metrics. This metric only

[1] OHE 1 autoencoder with compression ratio of 32

[2] OHE 1 autoencoder with compression ratio of 16

allows a comparison of models if applied to identical data. Image compression autoencoders prefer metrics like PSNR and compare them to classic compression algorithms. For example, Zebang and Sei-ichiro [112] use it to conclude that they achieve an 11% better performance than JPEG. When compressing scientific data consisting of floating-point numbers, Liu et al. [71] compress up to 2-4$\times$ better than SZ and 10-50$\times$ better than ZFP for a relatively large error bound of 0.1. These results can also not be compared to our works because no error bound is used for our models but an accuracy score to evaluate exact matches.

NLP models also use the accuracy score to evaluate their performance. Our SLE 1 autoencoder achieves an overall accuracy of 33.3% (*Header* 61.5%, *Cluster Centroids* 41.8%, *ADC values* 31.0%). And, our OHE 1 autoencoder achieves an overall accuracy of 25.0% (*Header* 100%, *Cluster Centroids* >95.6%, *ADC values* 0.0%). Compared to this, GPT-2 only answers 4.1% questions accurately when evaluated by an exact match metric. For the top 1% questions where GPT-2 feels most confident, it has an accuracy of 63.1%. GPT-3, the more advances version of GPT-2, has a larger performance increase. For paragraph and story completions, its accuracy is between 70 – 88%, translations to a foreign language are around 21 – 33% accurate, while the translation to English performs better, reaching an accuracy of 40%. When answering open questions, GPT-3 is 25 – 42% accurate, while for trivia questions it is 64 – 71% accurate. Based on these numbers, GPT-3 is on average about 15 – 20% better than GPT-2.

Comparing our models with the NLP models, we see that SLE 1 performs similarly to them. It has a model size as GPT-2, and they also have similar accuracy. Both have a maximum accuracy score of around 62% for the best data selection. The difference is in the number of exact matches. Here, GPT-2 is only 4.1% accurate, while SLE 1 has an accuracy of 33.3%. With an accuracy of 25%, OHE 1 performs like GPT-3 for complex tasks. However, we know that the large majority of OHE 1's errors are only found in the *Cluster Centroids*.

This comparison shows that our models have a mediocre performance compared to NLP models. Seeing the improvements of GPT-3 compared to GPT-2 makes us hopeful that such progress can also be achieved for our models in the future.

Conclusion

In this section, we have only explored and evaluated two different compression techniques that utilize domain knowledge, one lossless and one lossy. The Huffman Encoding using absolute values achieves the most promising performance, while the autoencoders in this particular hardware setup cannot be considered a practical solution. However, many compression algorithms are still left to explore that use domain knowledge. For example, one straightforward approach could be integrating the removal of events with no clusters. For LHCb 1, this could make up to 3% of the VELO file size. Another approach could be re-arranging the VELO data format to compress pairs of *Cluster Centroids* and their corresponding *ADC values*. Overall, our study underscores the beneficial usage of Huffman Coding. With this, it joins the ranks of many other works and (general-purpose) compression algorithms that use it.

Having now analyzed so many different compression techniques, the next chapter will perform a cost-benefit analysis to evaluate how economically viable each solution is for the deployment within the real-time LHCb DAQ.

Chapter 7

Cost-Benefit Analysis of Compression for LHCb

7.1 Introduction

The previous chapters proposed multiple ways to utilize data compression. In chapter 4, the performance of lossless general-purpose compression algorithms across ARM aarch64, IBM ppc64le, and Intel x86_64 architectures are analyzed. In chapter 5, a policy is proposed to efficiently and fairly co-schedule host and device workloads using a compression accelerator as an example. And in chapter 6, multiple compression algorithms are explored that take advantage of domain knowledge of the data.

With all these different solutions, the question arises as to which of them brings economic benefits when deployed as part of the LHCb DAQ. To evaluate this, we will look at two metrics: 1) capital expense and 2) power savings when deploying compression compared to the (non-compression) default setting.

We will first describe the LHCb DAQ setup and its requirements and limitations. Afterwards, we will perform the cost-benefit analysis for 1) general-purpose lossless compression algorithms, 2) commercial compression accelerators using the AHA378, and 3) compression algorithms utilizing domain knowledge: Huffman coding with delta encoding, and PCA-like autoencoders.

7.1.1 LHCb DAQ Run 3 Setup

The proposal for the DAQ for Run 3 of the LHCb experiment was planned as early as 2012 [23] for a realization in the years 2020-2021. By now, some changes have been introduced

Table 7.1 Servers

	Storage Xeon	HLT2 EPYC
Architecture	Intel	AMD
Platform	x86_64	x86_64
CPU Type	Gold 6342 @ 2.80GHz	EPYC 7302 @ 3.0 GHz
Release Date	2021	2019
TDP [W]	230	155
Virtual Cores	96	64
Threads per Real Core	2	2
Real Cores per Socket	24	16
Sockets	2	2
NUMA Nodes	2	2
RAM [GB]	256	480
Memory Channels per Socket	8	8
Max. Memory Bandwidth per Socket[GB/s]	204.8	204.8
PCIe	Gen4	Gen4
Idle Power Consumption [W] (Dual Socket)	160	149

compared to the original proposal, e.g., HLT1 is now solely run on GPUs instead of CPUs. Also, due to Covid-19, the realization phase is extended until 2022.

The current setup (Q1, 2022) of the DAQ for Run 3 of the LHCb experiment is shown in Figure 7.1. The compression should be deployed to compress data on the *HLT1 buffer*. The HLT1 buffer is between the so-called *EB-servers* and *HLT2 servers*. The EB-servers execute both assembling the subdetector data to a single event (event building) and afterwards filtering them using HLT1. The HLT1 buffer is managed by 16 *storage servers* which would perform the compression. They are connected to two JBOD arrays via (redundant) SAS cables. Each JBOD chassis supports up to 102 HDDs. The HLT2 farm consists of around 4000 x86_64 servers of various generations with a total of about 80,000 (real) cores. Table 7.1 lists the technical specifications of the newly acquired servers, both for storage and HLT2. And Table 7.2 lists the power consumption of HLT1 buffer The power consumption of the JBOD is measured using the software `fio` in direct I/O mode and random read over the entire HDD.[1] `Fio` is executed on a single storage server which for this purpose is only connected to a single JBOD. 102 `fio` instances are started - one for each HDD in the JBOD. The run

[1]Random read for HDD is a good approximation to max out the power consumption of the HDD, as Inoue et al. [51] and Andrew Binstock [13] show that read requires minimally more power per sec than write.

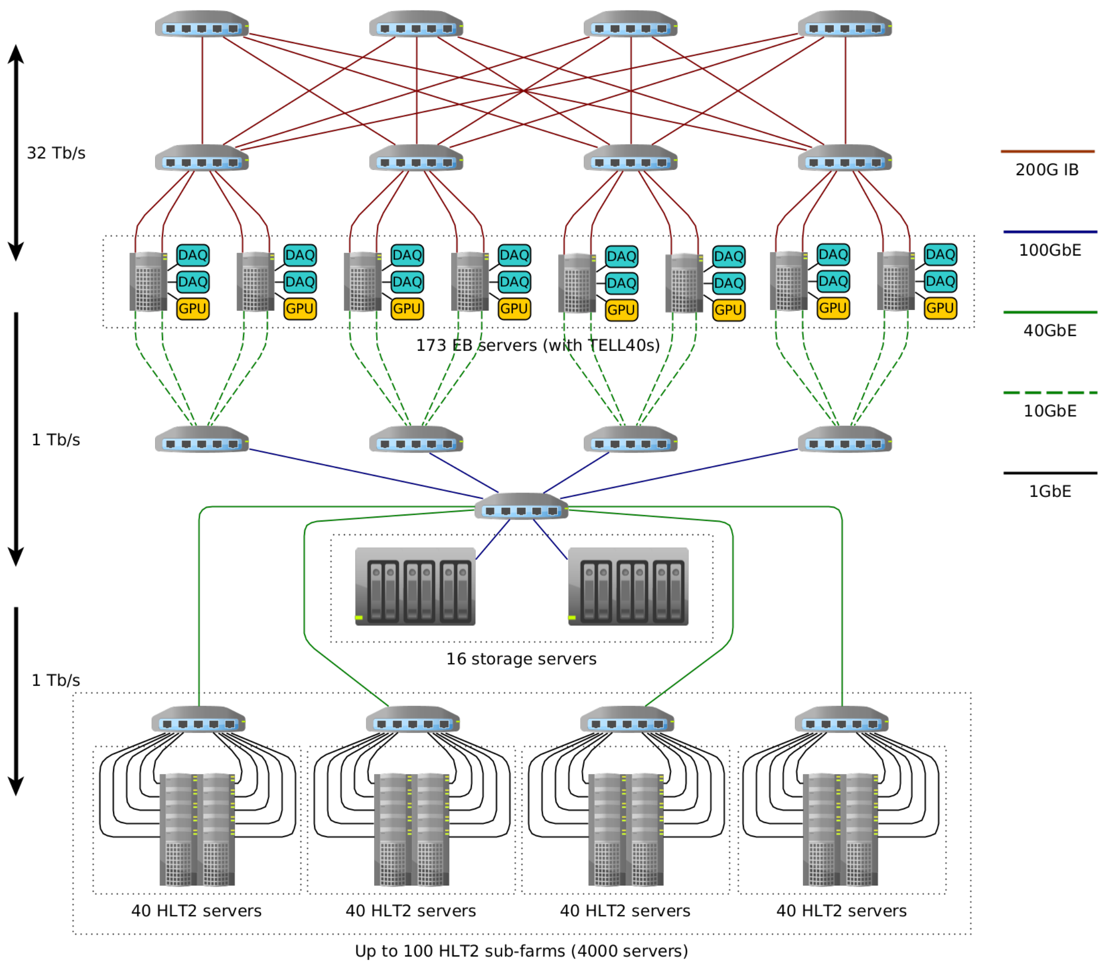


Fig. 7.1 LHCb DAQ for Run 3. The compression should be deployed on the 16 storage servers that manage the HLT1 buffer (HLT1 buffer not included in the graphic). [25]

time is 240 seconds to have a stable power consumption on the JBOD. It was not possible to measure the power consumption of a single HDD. Instead, the nominal power consumption by the manufacturer is used.

7.1.2 Test data

At the time of writing (March 2022), there is no LHCb Run 3 detector data available that could be used for the cost-benefit analysis. While simulated data is available, there is not enough data in the correct format available for our purpose. Most simulated data is in a format that can be directly digested by HLT2, while we need data in the format of how the detector front-ends would produce it. Therefore, we use Run 2 detector data and hope they will have similar compression characteristics due to the fact that the general structure of

Table 7.2 Power Consumption of Storage

JBOD	
HDDs per Chassis	102
Idle Power Consumption [W]	913
Active Power Consumption [W]	1309
HDD	
Size [TB]	14
Rotational Speed [RPM]	7200
Idle Power Consumption [W]	5.9
Active Power Consumption [W]	8.5

data stays the same for Run 3. Three files are used from different years. Two out of three of the files have a very similar compression ratio of 1.17 for `zlib level 2`, and one has a smaller compression ratio of 1.14. The difference in compression ratio can be pointed to two subdetectors with a higher-than-average compression ratio that contribute in different quantities to the data set sizes. In LHCb 2, the two subdetectors make up only 4.2% of the entire file, while for LHCb 1, it is 9.5%, and for LHCb 3, it is 10.5%.

7.1.3 Storage Servers, HLT1 Buffer, and Data Compression

The idea is to apply the data compression to the HLT1 buffer that stores pre-selected but unmodified data from the detector. The HLT1 buffer is controlled by the 16 storage servers. They use direct I/O to directly access the HDDs with no file system in between. The storage servers must provide an aggregated bidirectional I/O of 300 GB/s. This means 150 GB/s (1172 Gb/s) from HLT1 into the buffer and 150 GB/s out to HLT2. There are two primary motivators of the HLT1 buffer: 1) a necessary feedback loop to be able to realign the detector based on recently acquired data to measure more accurate data, and 2) the decoupling of real-time HLT1 and (none real-time) HLT2, allowing HLT2 to run while HLT1 has downtime (e.g., due to LHC maintenance). This lessens the requirement that HLT2 has to consume at the same speed as HLT1 produces. Compression for the HLT1 buffer would further allow us to either reduce the data transfer rates to and from the storage, increase the buffering capacity for the same amount of HDDs, or require fewer HDDs for the same buffering capacity.

For the compression, there are two possible configurations of deployment

1. Mono configuration: Compression and decompression on storage servers

2. Hybrid configuration: Compression on storage servers, decompression on HLT2 farm

Table 7.3 Data sets used for the cost evaluation

Name	Year	Size	Compression Ratio `zlib level 2`
LHCb 1	2016/17?	4.9 GB	1.16
LHCb 2	2017	4.6 GB	1.14
LHCb 3	2018	4.0 GB	1.17

The hybrid configuration would also include a reduction of network traffic between storage servers and the HLT2 farm.

7.1.4 Limitations of the Cost Estimation

Multiple factors limit the accuracy of this cost estimation. For example, at the current time of writing, the software is still missing for the bookkeeping of the data on the storage servers and the software for the HLT2 farm. Both are in development, but in their current state, they do not allow us to get a realistic impression of how well they can be co-scheduled with the different compression technologies. Therefore, this part is omitted in this analysis.

7.2 Using General-Purpose Compression Algorithms

In chapter 4 we evaluated eight different algorithms and compression levels: `bzip2 level 9`, `lz4 level 2`, `snappy`, `xz level 1` and `level 5`, `zlib level 2`, `zstd level 2` and `level 21`. While `xz level 5` and `zstd level 21` provide a very high compression ratio, their throughput is so low that we will not even consider them as a viable option. They are therefore omitted.

Selection of Algorithms: Throughput Performance

The compression will be solely executed on the storage servers. Therefore, the first step will be to evaluate if the compression algorithms are fast enough to reach the 150 GB/s (uncompressed) throughput on the 16 storage servers. We will ignore for now that each of those servers must run some form of bookkeeping and I/O scheduling software to interact with the JBODs. Table 7.4 lists the number of storage servers needed for compression. It turns out that `bzip2 level 9`, `xz level 1` and `zlib level 2` require many times over the available amount of storage servers and, as such, are not further evaluated.

Depending on the CPU resources, the decompression can happen either on the storage servers or on the HLT2 farm. Table 7.4 lists also those numbers. For HLT2, the numbers are

Table 7.4 Compression Algorithms: Amount of servers needed to fulfill the I/O requirements

Library	bzip2	lz4_HC	snappy	xz	zlib	zstd
Level	9	2	-	1	2	2
	Compression					
Throughput per Server [GB/s]	0.54	37.7	45.1	0.24	2.0	13.0
#Storage Servers	281	4	4	629	75	12
	Decompression on Storage server					
Throughput per Server [GB/s]	-	101.5	126.0	-	-	33.3
#Storage Servers	-	2	2	-	-	5
	Decompression on HLT2 server					
Throughput per Server [GB/s]	-	76.0	76.0	-	-	25.0
#HLT2 (real) cores	-	64	64	-	-	193

given as *cores* as the decompression should be performed by each server separately for the data they are processing. `Lz4 level 2` and `snappy`, being the algorithms with the lowest compression ratio (1.05 and 1.04) but also the highest throughput, needed the same total amount of 6 servers to perform both compression and decompression only on the storage servers. If decompression is executed on the HLT2 farm, they would need 64 (real) cores or 0.08% of the HLT2 farm. `Zstd level 2` would need 17 storage servers for compression and decompression. Therefore, this is not possible. However, it could be possible to use 12 storage servers for compression and 193 (real) cores or 0.24% of the HLT2 farm.

Cost-benefit Evaluation

Based on the previous throughput performance, we will perform the cost-benefit evaluation for `lz4 level 2` and `snappy` for both mono and hybrid configuration, and for `zstd level 2` for the hybrid configuration.

Table 7.5 lists this evaluation. In the table, section *Saved Resources* includes the JBODs saved, and the additional HDDs saved on top. The power consumption consists of the measured JBOD power consumption with all HDDs active, and for the additional HDDs, the active power consumption provided by the manufacturer. While it is not common to buy large quantities of JBODs that are only partly populated, it allows approximating the maximum possible savings when accounting for both the savings based on JBODs and the savings based on the additional HDDs.

The results show that neither `lz4 level 2` nor `snappy` should be used. Even though they provide high throughput, the compression ratio is too low to be beneficial for the overall calculations. The mono configuration performs in both cases worse than the hybrid

Table 7.5 Cost-benefit Evaluation: General-purpose Compression

	`Lz4 level 2`	`Snappy`	`Zstd level 2`
	Compression - Storage Server		
Compression Ratio	1.05	1.04	1.17
Storage Servers	4	4	12
Power Consumption [W]	1800	1692	5472
	Saved Resources - Storage		
JBODs	1	1	4
HDDs	53	23	66
Power Consumption [W]	1759	1504	5795
	Decompression - Storage Server		
Storage Servers	2	2	-
Power Consumption [W]	914	910	-
	Decompression - HLT2 Server		
HLT2 Cores	64	64	193
Power Consumption [W]	568	510	1894
	Total Savings - Mono Configuration		
Capital Expenses [%]	-21.7	-51.4	-
Power Consumption [%]	-54.3	-73.0	-
	Total Savings - Hybrid Configuration		
Capital Expenses [%]	-16.6	-9.3	9.6 $\Uparrow$
Power Consumption [%]	-55.7	-27.7	-15.8

configuration. While `snappy` has a lower compression ratio, it performs better than `lz4 level 2` in comparison to the initial savings of the storage infrastructure. Compared to the default setup, `snappy` needs for the hybrid configuration an additional 420 W (-27.7%) in power consumption, and it is 9.3% more expensive in capital expenses.

`Zstd level 2` performs overall the best. It is only evaluated for the hybrid configuration as the mono configurations need too many storage servers. It has a higher compression ratio, allowing it to save nearly $4\times$ as much in the storage infrastructure compared to `snappy`. Compared to the default setup, `zstd level 2` needs an additional 915 W in power consumption (-15.8%) and is the only algorithm that saves capital expenses. It is about 9.6% cheaper than the default setup. The power consumption does not consider the power savings due to the reduced amount of data transferred between the HLT1 buffer and HLT2 farm when using the hybrid configuration as the network cards stay active and do not power down during idle times. Hypothetically, using the specifications of Broadcom 10G NIC that consumes 7.9 W, the savings could be up to 300 W [14]. This would still result in an overall negative balance of about 600 W in power consumption. In addition, one problem which is difficult to solve is that `zstd level 2` needs 12 fully-occupied storage servers. At the same time, it is unlikely that the bookkeeping software for the storage will be able to perform well enough while using only a core count equivalent to four storage servers.[2]

Therefore, none of the proposed algorithms seem an economically viable choice for the LHCb DAQ.

7.3 Using Compression Accelerators

In chapter 5 we propose a policy that allows to efficiently and fairly co-schedule host and device workloads. The feasibility of using compression accelerators for the HLT1 buffer is being evaluated with the same AHA378 accelerator and its smaller counterpart, the AHA374[3]. The AHA374 uses the same FPGAs as the AHA378 but has only two instead of four. As such, it is a single-width PCIe card. While we have no physical access to an AHA374, it can be safely assumed that its performance will be at least half of the AHA378. This is derived from the technical specifications [45] published by the manufacturer for the AHA374 and our experiences with the performance of the AHA378 during this work. It uses the same FPGAs and needs more than half the power compared to the AHA378 while being advertised to achieve half of the throughput. Table 7.6 gives an overview of the accelerators and their (estimated) performance for the LHCb data sets.

[2]At the time of writing, no measurements are available to confirm or deny this.

[3]Other compression accelerators will be discussed in Chapter 8: Discussion.

Table 7.6 Accelerator Cards

	AHA374 (estimated)	AHA374 (measured)
Power Consumption [W]	40	60
PCIe Slots Width	Single	Dual
Compression ratio	1.17	1.17
	Stable Throughput [Gb/s]	
Design Throughput	40	80
Only Compression	38	78
Only Decompression	22	43
Both: Compression, Decompression	21,21	42,42

With the predetermined setup of the storage servers, many PCIe slots are already occupied by network cards to communicate with the JBODs, EB servers, and the HLT2 farm. There are only enough slots to either have three AHA374 accelerators, or a single AHA378 and a single AHA374 accelerator per storage server. For 16 storage servers, this would mean either 48 AHA374 accelerators, or 16 AHA378 and 16 AHA374 accelerators.

Configurations Based on Throughput Performance

Table 7.7 lists the amount of accelerators need for each configuration. Three of them turn out to be possible based on the hardware limitations. For the mono configuration, there is a single configuration. It would be a mixed configuration of 20 AHA374 and 20 AHA378 accelerators. However, they would need four additional storage servers to house them all. For the hybrid configuration, there are two possible configurations. Using 16 AHA378 accelerators, or 11 AHA374 and 11 AHA378 accelerators for compression. The decompression would always occur on the HLT2 farm, needing around 0.72% of the entire farm.

Cost-benefit Evaluation

Table 7.8 lists the cost-benefit evaluation of the three selected configurations. The mono configuration includes the additional costs of the four servers needed to house all accelerators. As in the previous section, the hybrid configuration does not consider the power savings due to the reduced network traffic.

All the configurations result in a net positive outcome. The mono configuration, using AHA374 and AHA378 for both compression and decompression, yields the best result in terms of energy efficiency. With a power-saving of 3135 W, it saves nearly twice as much power as the other configurations (1575 W and 1715 W). The savings in the capital

Table 7.7 Compression Accelerators: Amount of hardware needed to fulfill the I/O requirements

Configuration	AHA374	AHA378	HLT2 cores (only decomp)	Viability
	Mono Configuration			
Only AHA374	60	0	0	No
Only AHA378	0	30	0	No
Mixed	20	20	0	Needs +4 servers
	Hybrid Configuration			
Only AHA374	32	0	577	No
Only AHA378	0	16	577	Yes
Mixed	11	11	577	Yes

Table 7.8 Cost-benefit Evaluation: Accelerators

	Mono - Mixed	Hybrid - Only AHA378	Hybrid - Mixed
	Compression - Storage Server		
Compression Ratio	1.17	1.17	1.17
AHA374	10	0	11
AHA378	10	16	11
Power Consumption [W]	1000	960	1100
	Saved Resources - Storage		
JBODs	4	4	4
HDDs	66	66	66
Power Consumption [W]	5795	5795	5795
	Decompression - Storage Server		
AHA374	10	-	-
AHA378	10	-	-
Power Consumption [W]	1000		
	Decompression - HLT2 Server		
HLT2 Cores	-	577	577
Power Consumption [W]	-	3119	3119
	Total Savings		
Capital Expenses [%]	1.7	2.0	1.4
Power Consumption [%]	7.5	4.1	3.8

expenses are marginal in all three cases. Using a hybrid configuration does not turn out to be well-performing. The cost of power increases significantly, and the capital expenses are in the same range as for the mono configuration. Therefore, the mono configuration would be the best configuration when using compression accelerators. It saves minimally on capital expenses, but the better power profile will allow for long-term savings. The accommodation of the four additional servers is no problem.

7.4 Using Algorithms with Domain-Knowledge

Neither explored compression techniques using domain knowledge are compatible with the requirements of the LHCb DAQ.

The Huffman Encoding using the absolute values has by far the best compression ratio, but as it is slower than `bzip2 level 9`, it cannot be considered. We showed in section 7.2 that `bzip2 level 9` is too slow for the DAQ.

The autoencoders are no option either, as one of the requirements is that the compression must be lossless, but autoencoders compress lossy. This was known before selecting autoencoders as the second compression technique for domain knowledge.

7.5 Discussion

Ultimately, the only economically viable solution for the LHCb DAQ is using compression accelerators. But even those benefits are minimal when related to the entire compute hardware of the DAQ. This is unfortunate, especially since none of the domain knowledge compression techniques turn out to be beneficial.

However, the excellent compression results of the Huffman Encoding could be useful at other points in the pipeline. For example, it could be considered for the long-term storage of the raw detector data. Here, the compression performance should be of greater importance than the throughput. Especially when the data is replicated from CERN throughout the world. This is done for data redundancy and so that physicists can have a selection of multiple TB or even PB of data locally available to speed up their analyses significantly.

An additional, simple and fast compression technique could be integrated. With the characterization of the VELO data in the previous section, we learned that 42% of the VELO events contain zero-information data. This could save around 3% of all VELO data. The compression ratio of VELO is 1.11 and makes up around 12.3 – 13.5% of all the detector data. Applying these numbers would increase the total compression ratio by 0.4% to 1.1696 for LHCb 1 for little cost. It can be expected that also other sub-detectors have zero-information events that could be omitted. However, depending on the downstream implementations, software might need adaptions to handle events that exclude zero-information subdetectors.

Chapter 8

Discussion

We designed, characterized, and analyzed multiple different compression techniques during this work. In chapter 4, we analyzed the performance characteristics of general-purpose lossless compression algorithms across CPU architectures. In chapter 5 we developed a co-scheduling policy that fairly balances host and device workloads to allow the efficient use of compression accelerators, and in chapter 6 we explored two compression techniques that utilize domain knowledge: 1) Huffman and delta coding, and 2) PCA-like autoencoders. Finally, in chapter 7 we evaluated all previous compression techniques for their viability to be deployed in the real-time LHCb DAQ based on capital expenses and power savings.

General-Purpose Compression Algorithms

To characterize general-purpose lossless compression algorithms, we use a total of eight combinations of compression algorithms and levels. They are analyzed on three different server CPU architectures: ARM aarch64, IBM ppc64le, and Intel x86_64. The analysis concludes that the algorithms do not perform differently on the different architectures. Furthermore, the IBM and ARM servers are true to their heritage: IBM is the most powerful per core but also the most power-consuming, while ARM is the most power-efficient but also the weakest per core. The robustness and scalability are related to the maximum `SMT`. Having a high number of `SMT` results in a less robust but better-scaling performance. This means that the resource occupation of the compression algorithms is not optimal, as using a higher level of `SMT` results in a performance increase. The stalling in performance and loss of robustness of the IBM server shows that a single core becomes saturated when using 5-7 threads per core for the compression, while for ARM, it is between 3-4 threads per core.

While we have tested many algorithms and levels, only three of them are of practical use

- `snappy` for high throughput and low compression ratio

- `zstd level 2` (or the default: 3) for medium throughput and compression ratio
- `xz level 5` for low throughput and high compression ratio

`Snappy` is to be preferred over `lz4` as our measurements show that `snappy` compresses 20% faster, and decompresses 24% faster than `lz4`, while losing <2% of the already very low compression ratio. `Zstd` is a good replacement for `zlib`. It is a faster and equally well-compressing algorithm actively maintained by Facebook and the open source community.

While general-purpose compression algorithms are chosen for their long-term availability and maintenance by a third party or open source communities, this still does not prevent finding bugs years later. For example, at the end of March 2022, a severe `zlib`-bug was found. It allows memory corruption when decompressing data. This bug had a fix available since 2018, but the patch never reached people responsible for the `zlib` release [29].

Compression Accelerators

Communication becomes a significant factor for the overall performance when moving software parts from host to device. This is also the case when using compression accelerators. Therefore, we developed a policy that increases overall system performance by balancing the performance between host and device workloads fairly. It is robust to different host workload characteristics and requires no code changes for them. The policy's methodology uses well-known techniques, but they are applied outside the usual domain of network traffic and storage devices. Instead, they are used to design a generic policy that can be applied to any kind of co-hosted accelerator device. Especially as the policy does not require any code changes to the host workload, it is a compelling concept for integrating devices into existing systems.

The policy has the goal of covering all corner cases. Consequently, it is more complex in its decision flow than if, for example, it would not cover the case that host workloads can be both memory-heavy and compute-heavy or the case that a socket can consist of more than one NUMA domain. Using only the memory bandwidth as a metric would also simplify the decision flow but likewise degrade the performance.

We also executed the policy on the Intel E5-2650 v3 (same as in chapter 4), which is a dual-socket server with one NUMA domain per socket. The policy performed as expected. However, the results were not included in chapter 5 as during the policy evaluation, one of the memory slots (not memory modules), to which the accelerator was attached, broke. This could not be repaired, and to have reliable data, we performed a new evaluation on the AMD EPYC.

Other Accelerators While we based our policy only on the compression accelerator AHA378 from the AHA Products Group, we also gained experiences with three other commercial accelerators. Based on the data available to us, all of them had equal or worse compression performance than the AHA378.

The Zipline accelerator [98] originated from a collaboration of Broadcom and Microsoft. It is an NVMe-like device that implements the XP10 compression algorithm [77] and deflate. Direct contact was established with Broadcom in 2019 when the product was no longer exclusive to Microsoft but became commercially available. However, at that time, no Linux driver was available. Therefore, Broadcom themselves tested the performance on our data. They were surprised by the type of data we have as the compression ratio was only about 1.13 for XP10, while for their data, it outperforms `zlib level 9` [77] in compression ratio.

The NoLoad Computational Storage Processor of Eideticom Communications Inc. [50] is a commercial compression accelerator concept. Eideticom provides the implementation while a third party partner provides the FPGA hardware. These can be traditional PCIe cards or other shapes, like SSDs or HDDs. Compared to the AHA378, the performance seems more dependent on the input data.

For Broadcom and Eideticom, we had no access to the hardware but provided our data to their teams. However, we had one other commercial accelerator physically at hand: the Intel QuickAssist 8970 (QAT8970). The QAT8970 is an accelerator that provides functionality for cryptography and data compression. It is implemented in an ASIC and uses only 23 W. For cryptography, it offers industry standards for symmetric cryptography and public keys. For compression, only deflate is provided, but multiple performance levels are available to select. The communication with the host is similar to AHA378: a native API and a zlib-like wrapper called QATzip. The throughput provided by the card is up to 100 Gb/s for cryptography and 66 Gb/s for compression. In a more detailed benchmark document, Intel states for a 64 KB buffer size on Calgary corpus (the original 3 MB), the C267 Chipset (LGB-T), which should correspond to the QAT8970, achieves the following compression throughputs: 65 Gb/s for level 1, 35 Gb/s for level 2, 28 Gb/s for level 3, and 21 Gb/s for level 4. The compression ratios are not listed. Listed in Table 8.1 are our results for data sets Calgary corpus, LHCb 1, and CMS for accelerators QAT8970 and AHA378 executed on an Intel E5-2650 v3. In addition, results for `zstd level 2`, `zlib level 2` and `level 4` using `mtCPUComp` are provided to compare the accelerators performance with the CPU. The data set LHCb 1 has a low compression ratio common for HEP scientific data, Calgary corpus has a medium compression ratio common for text data, and CMS has an extremely high compression ratio. This is due to CMS consisting of test data that is mainly made up of zeros. Deflate is selected for both accelerators. In all cases compresses AHA378 better (up to 37%)

Table 8.1 Compression throughput and ratio for the data sets Calgary corpus, LHCb 1 and CMS for accelerators Intel QAT8970 and AHA378, and `zlib` and `zstd`.

Algorithm	Level		Calgary	LHCb 1	CMS
		Intel QAT8970			
deflate	1	Throughput	28.1	20.9	41.2
		Ratio	2.14	1.16	502
	2	Throughput	20.1	17.8	41.2
		Ratio	2.3	1.17	420
	3	Throughput	15.9	16.5	41.1
		Ratio	2.35	1.17	424
	4	Throughput	12.2	15.4	41.3
		Ratio	2.38	1.17	424
		AHA378			
deflate		Throughput	81.4	80.9	81.2
		Ratio	2.87	1.19	687
		`mtCPUComp`			
zlib	2	Throughput	29.8	15.2	146
		Ratio	2.74	1.17	205
	4	Throughput	21.1	13.3	77.3
		Ratio	2.93	1.18	724
zstd	2	Throughput	71.1	74.5	722
		Ratio	2.94	1.16	2732

and faster (up to 3.9×) than QAT8970. The QAT8970 results are significantly lower (42 - 57%) than presented by Intel. The compression ratio of both accelerators is between `zlib level 2` and `level 4`. `Zstd level 2` is with equal or better compression performance 2.3 – 4.9× faster than `zlib`. For high-compressing data input, the `mtCPUComp` algorithms outperform the accelerators. In all cases, `zstd level 2` outperforms the QAT8970. It was unexpected to see that the throughput performance of the QAT8970 does not reach anywhere close to Intel's official numbers and that with its data-dependent throughput, it behaves more like the CPU algorithms and not like the AHA378.

Overall, the QAT8970 underperformed in compression ratio, throughput, and usability[1]. While we have not tested the cryptography performance of QAT8970, it seems to be better supported, and one could hope that it performs better than the compression unit. Because we have with the AHA378 a compression accelerator that provides stable and high throughput performance, no further studies were conducted with the QAT8970, and the AHA378 was used to design the policy.

[1]Contact with Intel engineers could not improve the performance.

Compression with Domain Knowledge

Two compression techniques and their variations are selected to be evaluated using raw detector data of the subdetector VELO. The selection of the subdetector was based on being representative and having a small and simple data format to be able to focus on the compression techniques and not the implementation details. We first characterized the data. It concludes that the VELO fields have various distributions. To limit the complexity, techniques are selected that are either relatively unaffected by the underlying distribution or can automatically extract and train on the distributions.

We utilize Huffman coding 1) on delta encoded, and 2) on unmodified *absolute value* data. While it was expected that delta encoded data would perform better, the opposite is true. Huffman coding on absolute values achieves the best compression ratio and compresses 40 – 260% better than all tested algorithms and is 13% faster than Huffman coding on delta encoded data. This is explained by the delta encoded data (before Huffman coding) having a less steep, more homogeneous distribution of usage frequency of its values compared to the distribution of the absolute values. This diminishes the performance of the Huffman coding that assigns symbols with the highest usage frequency to the shortest code words.

To explore autoencoders, an autoencoder architecture is chosen that fulfills the mathematical requirements of a PCA. PCA is a widely-used transformation that reduces dimensionality while clustering the most important information in the first few dimensions. This reduces information loss and, as such, is ideal for compression. We have autoencoders for two different data representations, SLE and OHE. Many hyperparameters are evaluated to find the best configurations. While OHE can reconstruct the header without any errors, the overall performance of both SLE and OHE autoencoders is promising but still too lossy to be considered for LHCb DAQ. OHE performs well considering how few samples it is trained on (not even 1%) but requires large amounts of memory. Knowing in advance how much memory is precisely needed is difficult to estimate. The OHE 1 autoencoder uses around 260 GB RAM with peaks of up to 500 GB RAM during runtime. Of these 260 GB, 64 GB are already used just for the weights of all the layers. Improving this model requires an additional unrealistic amount of training on high-end CPUs on a single server, or it needs to be distributed onto multiple GPUs. If the GPU-approach is not possible, technology must advance further to improve the monolithic training of such large networks. Comparing our autoencoders to other works shows that the other works might have more parameters in total, but the layer sizes are smaller to improve training time, general handling of model and data, and their computation in distributed, heterogeneous environments.

At the moment, our results conclude that autoencoders are still better used in environments that accept lossy compression and have smaller layer sizes. For example, in image or video compression, they achieve good results. Even applied to floating-point numbers, they are more appropriate. For example, Liu et al. [71] produce remarkable results that outperformed classic compression techniques by a factor of 2 – 50. Floating-point numbers are, by implementation design, a lossy representation on CPU architectures - and as such, are quantized values. With this ideal for other lossy compression techniques as in most use cases only a fraction of its domain size is used.

Overall, when using compression techniques that integrate domain knowledge, it is challenging to find well-performing techniques for your data immediately, and you might get surprises which technique performs well. Undeniably, more research is needed to find better-performing techniques for our data. But this is expensive both in time and cost, as analysis and custom implementations are needed.

Cost-Benefit Analysis

During the cost-benefit analysis, we saw how difficult it is to find a suitable compression candidate. The real-time requirements of the LHCb DAQ, with its high throughput and a limited amount of servers usable for compression, severely limit the number of possible candidates. The general-purpose compression algorithms have no suitable candidates. This was expected as compressing on the CPU is slow, and the general-purpose compression algorithms are all designed to perform well on text data and not on scientific data. The interesting part is that their performance does not depend on the underlying CPU architecture. The compression algorithms are therefore not the decisive factor for the system hardware.

Using compression accelerators is economically feasible. More than one configuration results in a net-positive outcome for both power and capital expenses. However, the savings are not gigantic. These savings would significantly impact big companies like Google or Facebook. But for the LHCb DAQ, the savings are small when compared to the hardware costs of the entire DAQ pipeline. Considering that this DAQ setup will run for at least 10 years with no possibility of significant changes to the system design, another analysis would be beneficial that evaluates the long-term maintenance cost, including spare parts acquisition. The majority of the DAQ relies on commercial products available in large quantities, while the AHA products are commercially available but do not have a large market share.

For the compression with domain knowledge, we never considered autoencoders a possibility for the Run 3 DAQ as lossy compression is generally not accepted by the physicists. If physicists accept lossy compression, it would be a long process. The question of how much lossy compression is acceptable would be based on the HLT1 and HLT2 collision reconstruction performance based on the decompressed data provided by the autoencoder. Also, lossy compression must only be applied to the payload and not to the headers. Otherwise, assigning an event to its original particle collision would become impossible. Even though autoencoders compress lossy, the performance of the OHE autoencoders promises that with technological advances it can become a practical solution.

Huffman coding on absolute values is a compression technique utilizing domain knowledge that outperforms all other algorithms in compression ratio. While it is too slow for the LHCb DAQ, it should not be overlooked. It is a choice definitely worth investigating for locations that need a high compression ratio and accept a somewhat slow throughput. In our case, Huffman coding on absolute values is faster than `xz level 1` and compresses 40% better than `xz level 5`.

Generalization

While the background of this entire work is DAQs in HEP, the results are generic enough to be relevant for many different environments and data characteristics. The performance characteristics of general-purpose lossless compression algorithms are independent of the CPU architecture. While they perform best for text-based data input, they can be applied to any type of data. The choice of algorithms depends here on compression and throughput performance requirements.

The co-scheduling policy is a generic policy due to the fact that it is only interested in the performance metrics that influence the CPU resources. It is unperturbed by the exact choice of both host workload and device workload. The accelerator device can perform any other tasks, not limited to compression, as long as it supports both communication schemata: polling and interrupts. For the CPU, only the memory bandwidth usage and CPU power consumption are of interest. Thus, the policy is unaffected by the exact nature of the workloads.

Comparing the compression and decompression performance, we find, as expected, that decompression using general-purpose compression algorithms is generally much faster on the CPU than compression. However, this does not hold true for accelerators. Based on our data, the accelerators tend to compress faster than decompress. Especially, the decompression of HEP data is penalized in its throughput, with AHA378 just achieving 50% of its compression throughput.

While the compression with domain knowledge is tuned explicitly for HEP raw detector data, the general results should be valid anywhere: 1) there is always a better performing compression algorithm using domain knowledge compared to generic solutions, 2) it is expensive to find the best fitting technique, 3) autoencoders currently do not perform well enough for networks with large layer sizes and lossless compression requirements, and 4) OHE generally works better for discrete data but runs quickly into memory limitations.

Furthermore, not only HEP data is used, but also standard compression corpora. This allows comparing the results of this work with other compression research. For all these reasons, we firmly believe that this work contains generic results that can be applied to many different areas.

Chapter 9

Conclusion

In this work, we shed light on the trade-offs between compute, storage, and throughput and the implications on the system design when deploying data compression; the advantages and disadvantages of using general-purpose compression algorithms; how to efficiently and fairly integrate (compression) accelerators; and if using domain knowledge can improve the compression performance. This is all done in the context of real-time Data Acquisition Systems (DAQs) in the High Energy Physics (HEP) community which have specific throughput and storage requirements and constraints.

General-Purpose Compression Algorithms

The popular general-purpose lossless compression algorithms `bzip2`, `lz4`, `snappy`, `zlib`, `zstd` and `xz` are characterized across CPU platforms. One server for each CPU platform: ARM aarch64, Intel x86_64, and IBM ppc64le. The characterization includes the metrics: cross-platform behavior, performance per core (unified metric across the platforms), robustness, scalability, and power consumption and efficiency. They allow us to conclude that algorithms are not influenced by the underlying CPU architecture but depend on their performance on the input data. In addition, having a higher number of threads per core (SMTx) results in a less robust but better-scaling performance. When it comes to "green" computing, ARM outperforms IBM by a factor of 2.8 and Intel by 1.3.

From a system integration point of view, the three architectures perform equally, and as such, the choice of the architecture should be based on the cost-benefit analysis and the requirements for the rest of the system.

Co-scheduling Policy and Compression Accelerators

We developed a policy that provides an efficient and fair co-scheduling performance between independent host and (compression) device workloads. This policy requires no in-code changes to the host workload. It is based on two readily available performance metrics: memory bandwidth and CPU power consumption, and works by using NUMA-binding and selecting either polling or interrupts as communication schema for the device workload.

The policy always increases performance for simple cases where host workloads are either memory-heavy or compute-heavy. There, the host workload increases in performance by a factor of 1.8 – 2.3, while the device performance increases by 1.5 – 4.0. For complex cases, where host workloads are memory- and compute-heavy, the policy has a significant performance increase for the device by a factor of 1.8 – 1.9, while the performance of the host workload has a decrease of 0.1 – 0.4. Overall, the policy achieves good results with little effort for various workload characteristics. However, the limitations are evident: host workloads with high compute intensity, and high memory bandwidth make it challenging to optimize and fairly share the system performance.

In addition to the compression accelerator used to design this policy (AHA378), we compared the performance of three other commercial compression accelerators. Two, the Microsoft and Broadcom collaboration Zipline and Eideticom's NoLoad, we had no physical access to, but for the third, Intel QAT8970, we had. Based on the performance data we acquired, all three accelerators perform worse than the AHA378 in compression ratio - but also in some cases in throughput.

Compression using Domain-Knowledge

Two different compression techniques are evaluated: lossless Huffman coding on two data representations and multiple versions of lossy PCA-like autoencoders.

Huffman coding on *absolute values* outperforms *delta encoded* data and also all previously presented general-purpose compression algorithms by 40 – 260% in compression performance. In its throughput, it is 13% faster than the *delta encoded* data or right in the middle of the throughput of `bzip2 level 9` and `xz level 1`.

The autoencoders are trained on two data representations, *padded* (SLE) and *one hot encoded* (OHE), and multiple hyperparameters. The data is based on the LHCb raw detector data of the VELO subdetector that can be split into three parts: Header, Cluster Centroids, and ADC values. SLE achieves the best performance with the hyperparameters Adam, MSE, and ReLU. OHE achieves the best with Nadam, Huber or Log Cosh, and ReLU. Both autoencoders result in lossy compression. SLE has its best reconstruction results for Header and Cluster Centroids and worst for ADC values. OHE has perfect reconstruction for Header, small error for ADC values, and cannot reconstruct Cluster Centroids. In addition, it suffers from memory limitations resulting in no GPU training and only being trained on 1% of the data.

Cost-Benefit Analysis of Compression for LHCb DAQ

The economical cost-benefit analysis of integrating different compression techniques into the real-time LHCb DAQ is based on power savings and capital expenses. For the general-purposes compression algorithms, only `snappy` and `lz4 level 2` would satisfy the hardware requirements of supporting the 150 GB/s on a portion of the 16 storage servers. Still, power savings and capital expenses result in a net-negative evaluation. As such, no general-purpose compression algorithm is economical. For compression accelerators, three different solutions result in a net-positive outcome. Huffman coding on absolute values achieves the best compression ratio but has the same problem as most general-purpose compression algorithms: it is too slow. Autoencoders are not considered as they compress lossy.

Ultimately, using compression accelerators is the only solution that fulfills all requirements and constraints of the LHCb DAQ.

Closing remarks

Overall, we show in this work that the trade-offs between compute, storage, and throughput is a complicated endeavor for real-time DAQs. The final result of this work is that only compression accelerators can fulfill the requirements unless a lot of resources are invested in finding the optimal compression technique utilizing domain knowledge. Commercial compression accelerators have the speed advantage and can meet the throughput requirements most cost-efficiently. Their `deflate` implementation reaches a medium compression ratio and can be decompressed with any `zlib` implementation. Furthermore, they have the advantage of being able to be bought off the shelf.

Even though this work focuses on real-time DAQs in the HEP community, which have specific throughput and storage requirements and constraints, we firmly believe that the results of this work are relevant and can be generalized to many different environments and data characteristics.

Chapter 10

My Publications

Related to This Work

[89] Laura Promberger, Rainer Schwemmer, and Holger Fröning. Assessing the Overhead of Offloading Compression Tasks. In *49th International Conference on Parallel Processing - ICPP: Workshops*, ICPP Workshops '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450388689. doi: 10.1145/3409390.3409405. URL https://doi.org/10.1145/3409390.3409405
[90] Laura Promberger, Rainer Schwemmer, and Holger Fröning. Characterization of data compression across CPU platforms and accelerators. *Concurrency and Computation: Practice and Experience*, page e6465, 2021. doi: https://doi.org/10.1002/cpe.6465. URL https://misclibrary.wiley.com/doi/abs/10.1002/cpe.6465
[91] Laura Promberger, Holger Fröning, and Rainer Schwemmer. Characterization and Integration of Accelerated Compression for a Fair System Performance. *Submitted for Publication*, 2022

Unrelated to This Work

[88] Laura Promberger, Marco Clemencic, Ben Couturier, Aritz Brosa Iartza, and Niko Neufeld. Porting the LHCb Stack from x86 (Intel) to aarch64 (ARM) and ppc64le (PowerPC). In *EPJ Web of Conferences*, volume 214, page 05016. EDP Sciences, 2019

References

[1] Roel Aaij, Daniel Hugo Cámpora Pérez, Tommaso Colombo, Conor Fitzpatrick, Vladimir Vava Gligorov, Arthur Hennequin, Niko Neufeld, Niklas Nolte, Rainer Schwemmer, and Dorothea Vom Bruch. Evolution of the energy efficiency of LHCb's real-time processing. *EPJ Web of Conferences*, 251:04009, 2021. ISSN 2100-014X. doi: 10.1051/epjconf/202125104009. URL http://dx.doi.org/10.1051/epjconf/202125104009.

[2] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B. Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J. Starke, Haren Myneni, and Charlie Wang. Data Compression Accelerator on IBM POWER9 and z15 Processors : Industrial Product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2020. doi: 10.1109/ISCA45697.2020.00012.

[3] Mohamed S. Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a Chip: High Performance Lossless Data Compression on FPGAs Using OpenCL. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, IWOCL '14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330077. doi: 10.1145/2664666.2664670. URL https://doi.org/10.1145/2664666.2664670.

[4] Dr.-Ing. Jürgen Abel. Corpora, Last accessed: January 18, 2022. URL http://www.data-compression.info/Corpora/index.html.

[5] Mark Adler and Greg Roelofs. zlib Home Site, Last accessed: August 2, 2019. URL https://www.zlib.net/.

[6] Jyrki Alakuijala, Andrea Farruggia, Paolo Ferragina, Eugene Kliuchnikov, Robert Obryk, Zoltan Szabadka, and Lode Vandevenne. Brotli: A General-Purpose Data Compressor. *ACM Trans. Inf. Syst.*, 37(1), dec 2018. ISSN 1046-8188. doi: 10.1145/3231935. URL https://doi.org/10.1145/3231935.

[7] AMD. AMD μProf - AMD, Last accessed: Accessed on 25th August, 2020. URL https://developer.amd.com/amd-uprof/.

[8] S. Amiri, M. Hosseinabady, A. Rodriguez, R. Asenjo, A. Navarro, and J. Nunez-Yanez. Workload Partitioning Strategy for Improved Parallelism on FPGA-CPU Heterogeneous Chips. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 376–3764, 2018.

[9] SuSE Labs Andi Kleen. numactl(8) - Linux man page, Last accessed: March 30, 2022. URL https://linux.die.net/man/8/numactl.

[10] Naiyong Ao, Fan Zhang, Di Wu, Douglas S. Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. Efficient Parallel Lists Intersection and Index Compression Algorithms Using Graphics Processing Units. *Proc. VLDB Endow.*, 4(8):470–481, May 2011. ISSN 2150-8097. doi: 10.14778/2002974.2002975. URL http://dx.doi.org/10.14778/2002974.2002975.

[11] Aashish Barnwal. Efficient Huffman Coding for Sorted Input | Greedy Algo-4 - GeeksforGeeks, Last accessed: April 25, 2022. URL https://www.geeksforgeeks.org/efficient-huffman-coding-for-sorted-input-greedy-algo-4/.

[12] M. Belwal, M. Purnaprajna, and Sudarshan TSB. Enabling seamless execution on hybrid CPU/FPGA systems: Challenges directions. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2015.

[13] Andrew Binstock. Newer Technology: News Room : Press Article : Measuring HDD Power Usage with Newer Tech Universal Adapter By Andrew Binstock, from Greener Computing, Last accessed: Accessed on 15th March, 2022. URL https://www.newertech.com/Static/articles/article_greenercomp_hhdpower.php.

[14] Broadcom. P210P - 2 x 10GbE PCIe NIC, Last accessed: April 8, 2022. URL https://www.broadcom.com/products/ethernet-connectivity/network-adapters/10gb-nic-ocp/p210p.

[15] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[16] Elizabeth Van Campen. Overview of loss functions for Machine Learning | by Elizabeth Van Campen | Analytics Vidhya | Medium, Last accessed: May 8, 2022. URL https://medium.com/analytics-vidhya/overview-of-loss-functions-for-machine-learning-61829095fa8a.

[17] Julián L Cárdenas-Barrera and Juan Valentin Lorenzo-Ginori. Mean-shape vector quantizer for ECG signal compression. *IEEE Transactions on Biomedical Engineering*, 46(1):62–70, 1999.

[18] CERN. CERN Annual report 2020 - CERN Document Server, Last accessed: January 25, 2022. URL https://cds.cern.ch/record/2771424?ln=en.

[19] CERN. Home | CERN, Last accessed: October 2, 2020. URL https://home.cern/.

[20] D. Chen and D. Singh. Fractal video compression in OpenCL: An evaluation of CPUs, GPUs, and FPGAs as acceleration platforms. In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 297–304, Jan 2013. doi: 10.1109/ASPDAC.2013.6509612.

[21] Zhengxue Cheng, Heming Sun, Masaru Takeuchi, and Jiro Katto. Deep convolutional autoencoder-based lossy image compression. In *2018 Picture Coding Symposium (PCS)*, pages 253–257. IEEE, 2018.

[22] Jee Whan Choi, Daniel Bedard, Robert Fowler, and Richard Vuduc. A roofline model of energy. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 661–672. IEEE, 2013.

[23] LHCb Collaboration. Framework TDR for the LHCb Upgrade: Technical Design Report, Apr 2012. URL https://cds.cern.ch/record/1443882.

[24] Lasse Collin. XZ Utils, Last accessed: August 2, 2019. URL https://tukaani.org/xz/.

[25] Tommaso Colombo. dedicated_eb.gpu.v3.png, Received on: February 4, 2022.

[26] J. Cong, Z. Fang, M. Huang, L. Wang, and D. Wu. CPU-FPGA Coscheduling for Big Data Applications. *IEEE Design Test*, 35(1):16–22, 2018.

[27] Intel Corporation. Intel QuickAssist Adapter 8970 Product Specifications, Last accessed: January 10, 2022. URL https://ark.intel.com/content/www/us/en/ark/products/125200/intel-quickassist-adapter-8970.html.

[28] Intel Corporation. Intel Performance Counter Monitor - A Better Way to Measure CPU Utilization, Last accessed: January 11, 2022. URL https://www.intel.com/software/pcm.

[29] GitHub Advisory Database. zlib 1.2.11 allows memory corruption when deflating (i.e.... · CVE-2018-25032 · GitHub Advisory Database · GitHub, Last accessed: April 22, 2022. URL https://github.com/advisories/GHSA-jc36-42cf-vqwj.

[30] L. Peter Deutsch. RFC 1951 - DEFLATE Compressed Data Format Specification version 1.3, Last accessed: March 9, 2020. URL https://tools.ietf.org/html/rfc1951.

[31] Hariharan Devarajan, Anthony Kougkas, and Xian-He Sun. An Intelligent, Adaptive, and Flexible Data Compression Framework. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 82–91, 2019. doi: 10.1109/CCGRID.2019.00019.

[32] Constantinos Dovrolis, Brad Thayer, and Parameswaran Ramanathan. HIP: Hybrid Interrupt-Polling for the Network Interface. *SIGOPS Oper. Syst. Rev.*, 35(4):50–60, October 2001. ISSN 0163-5980. doi: 10.1145/506084.506089. URL https://doi.org/10.1145/506084.506089.

[33] Timothy Dozat. Incorporating nesterov momentum into adam. 2016.

[34] Doris Eckstein. VELO Raw Data Format and Strip Numbering, Last accessed: April 25, 2022. URL https://edms.cern.ch/document/637676/2.

[35] Ramez Moh Elaskary, Mohamed Saeed, Tawfik Ismail, Hassan Mostafa, and Salam Gabran. Hybrid DCT/Quantized Huffman compression for electroencephalography data. In *2017 Japan-Africa conference on electronics, communications and computers (JAC-ECC)*, pages 111–114. IEEE, 2017.

[36] L. Erdodi and L. Erdődi. File compression with LZO algorithm using NVIDIA CUDA architecture. In *2012 4th IEEE International Symposium on Logistics and Industrial Informatics*, pages 251–254, 2012. doi: 10.1109/LINDI.2012.6319497.

[37] P. Ferragina and G. Navarro. Pizza&Chili Corpus – Compressed Indexes and their Testbeds, Last accessed: March 10, 2020. URL http://pizzachili.dcc.uchile.cl/texts/protein/.

[38] J. Fowers, J. 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*, pages 52–59, May 2015. doi: 10.1109/FCCM.2015.46.

[39] Antoine Le Gall. New 3D colour X-rays made possible with CERN technology | CERN, Last accessed: January 25, 2022. URL https://home.cern/news/news/knowledge-sharing/new-3d-colour-x-rays-made-possible-cern-technology.

[40] Mark A Goldberg, Misha Pivovarov, William W Mayo-Smith, Meenakshi P Bhalla, Johan G Blickman, Robert T Bramson, GW Boland, HJ Llewellyn, and Elkan Halpern. Application of wavelet compression to digitized radiographs. *AJR. American journal of roentgenology*, 163(2):463–468, 1994.

[41] Adam Golinski, Reza Pourreza, Yang Yang, Guillaume Sautiere, and Taco S. Cohen. Feedback Recurrent Autoencoder for Video Compression. In *Proceedings of the Asian Conference on Computer Vision (ACCV)*, November 2020.

[42] Ruan Delgado Gomes, Yuri Gonzaga Gonçalves da Costa, Lucenildo Lins Aquino Júnior, Manoel Gomes da Silva Neto, Alexandre Nóbrega Duarte, and Guido Lemos de Souza Filho. A Solution for Transmitting and Displaying UHD 3D Raw Videos Using Lossless Compression. In *Proceedings of the 19th Brazilian Symposium on Multimedia and the Web*, WebMedia '13, pages 173–176, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 978-1-4503-2559-2. doi: 10.1145/2526188.2526228. URL http://doi.acm.org/10.1145/2526188.2526228.

[43] google/snappy. GitHub - google/snappy: A fast compressor/decompressor, Last accessed: August 2, 2019. URL https://github.com/google/snappy.

[44] AHA Products Group. AHA Products Group, Last accessed: January 10, 2022. URL http://www.aha.com/DrawProducts.aspx?Action=GetProductDetails&ProductID=41.

[45] AHA Products Group. AHA Products Group, Last accessed: March 29, 2022. URL http://www.aha.com/Uploads/aha374-378_brief_rev_c1.pdf.

[46] AHA Products Group. AHA Products Group, Last accessed: October 25, 2019. URL http://www.aha.com/.

[47] G. Haefeli, A. Bay, A. Gong, H. Gong, M. Muecke, N. Neufeld, and O. Schneider. The LHCb DAQ interface board TELL1. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 560(2):494 – 502, 2006. ISSN 0168-9002. doi: https://doi.org/10.1016/j.nima.2005.12.212. URL http://www.sciencedirect.com/science/article/pii/S0168900205026100.

[48] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2017.

[49] Peter J Huber. Robust estimation of a location parameter. In *Breakthroughs in statistics*, pages 492–518. Springer, 1992.

[50] Eidetic Communications Inc. Products - Eideticom, Last accessed: April 25, 2022. URL https://www.eideticom.com/products.html.

[51] Takuro Inoue, Makoto Ikeda, Tomoya Enokido, Ailixier Aikebaier, and Makoto Takizawa. A Power Consumption Model for Storage-based Applications. In *2011 International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 612–617, 2011. doi: 10.1109/CISIS.2011.101.

[52] Intel. Intel Memory Latency Checker v3., Last accessed: Accessed on 24th August, 2020. URL https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html.

[53] Intel. Intel QuickAssist Technology (Intel QAT) Improves Data Center..., Last accessed: October 25, 2019. URL https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html.

[54] Tanzila Islam, Chyon Kim, Hiroyoshi Iwata, H. Shimono, Akio Kimura, Hein Zaw, Chitra Raghavan, Hei Leung, and Rakesh Singh. A Deep Learning Method to Impute Missing Values and Compress Genome-wide Polymorphism Data in Rice. pages 101–109, 01 2021. doi: 10.5220/0010233901010109.

[55] Jseward. bzip2 : Home, Last accessed: March 9, 2020. URL https://www.sourceware.org/bzip2/.

[56] Łukasz Kaiser and Samy Bengio. Discrete Autoencoders for Sequence Models, 2018. URL https://arxiv.org/abs/1801.09797.

[57] Keras. Keras: the Python deep learning API, Last accessed: April 22, 2022. URL https://keras.io/.

[58] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[59] Jana Kosecka. face-recognition.ppt, Last accessed: April 25, 2022. URL https://cs.gmu.edu/~kosecka/cs687/face-recognition.pdf.

[60] Anil Krishna, Timothy Heil, Nicholas Lindberg, Farnaz Toussi, and Steven VanderWiel. Hardware Acceleration in the IBM PowerEN Processor: Architecture and Performance. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 389–400, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1182-3. doi: 10.1145/2370816.2370872. URL http://doi.acm.org/10.1145/2370816.2370872.

[61] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[62] Koen Langendoen, John Romein, Raoul Bhoedjang, and Henri Bal. Integrating polling, interrupts, and thread management. In *Proceedings of 6th Symposium on the Frontiers of Massively Parallel Computation (Frontiers' 96)*, pages 13–22. IEEE, 1996.

[63] Dr. Cher Han Lau. >5 Steps of a Data Science Project Lifecycle | by Dr. Cher Han Lau | Towards Data Science, Last accessed: January 10, 2022. URL https://towardsdatascience.com/5-steps-of-a-data-science-project-lifecycle-26c50372b492.

[64] Duncan Laurie. GitHub - ipmitool/ipmitool: An open-source tool for controlling IPMI-enabled systems, Last accessed: April 8, 2022. URL https://github.com/ipmitool/ipmitool.

[65] Yann LeCun. The MNIST database of handwritten digits, 1998. URL http://yann.lecun.com/exdb/mnist/.

[66] S. Lee, J. Park, K. Fleming, Arvind, and J. Kim. Improving performance and lifetime of solid-state drives using hardware-accelerated compression. *IEEE Transactions on Consumer Electronics*, 57(4):1732–1739, November 2011. doi: 10.1109/TCE.2011.6131148.

[67] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. SIMD Compression and the Intersection of Sorted Integers. *Softw. Pract. Exper.*, 46(6):723–749, June 2016. ISSN 0038-0644. doi: 10.1002/spe.2326. URL https://doi.org/10.1002/spe.2326.

[68] LHCb. b is for beauty, Last accessed: January 25, 2022. URL https://lhcb-outreach.web.cern.ch/physics/b-is-for-beauty/.

[69] LHCb. Collaboration, Last accessed: January 25, 2022. URL https://lhcb-outreach.web.cern.ch/collaboration/.

[70] Rolf Lindner. LHCb-upgrade-y-no-logo.jpg, Received on: February 7, 2022.

[71] Tong Liu, Jinzhen Wang, Qing Liu, Shakeel Alibhai, Tao Lu, and Xubin He. High-ratio lossy compression: Exploring the autoencoder to compress scientific data. *IEEE Transactions on Big Data*, 2021.

[72] Lz4. LZ4 - Extremely fast compression, Last accessed: August 2, 2019. URL http://www.lz4.org.

[73] Matt Mahoney. About the Test Data, Last accessed: March 10, 2020. URL http://mattmahoney.net/dc/textdata.html.

[74] G. S. Martins, D. Portugal, and R. P. Rocha. A comparison of general-purpose FOSS compression techniques for efficient communication in cooperative multi-robot tasks. In *2014 11th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, volume 02, pages 136–147. INSTICC, Sep. 2014. doi: 10.5220/0005058601360147.

[75] J. Matai, J. Kim, and R. Kastner. Energy efficient canonical huffman encoding. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 202–209. IEEE Computer Society, June 2014. doi: 10.1109/ASAP.2014.6868663.

[76] Joris Mertens. IBM Z fundamentals: An introductory Q&A - IBM Developer, Last accessed: January 12, 2022. URL https://developer.ibm.com/articles/what-is-ibm-z/.

[77] michaelgmcintyre. Project-Zipline/FAQ.md at master · opencomputeproject/Project-Zipline · GitHub, Last accessed: April 25, 2022. URL https://github.com/opencomputeproject/Project-Zipline/blob/master/FAQ.md.

[78] A. Milenkovic, A. Dzhagaryan, and M. Burtscher. Performance and Energy Consumption of Lossless Compression/Decompression Utilities on Mobile Computing Platforms. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 254–263. IEEE Computer Society, Aug 2013. doi: 10.1109/MASCOTS.2013.33.

[79] Esma Mobs. The CERN accelerator complex - 2019. Complexe des accélérateurs du CERN - 2019, Last accessed: April 8, 2022. URL https://cds.cern.ch/record/2771424?ln=en.

[80] J. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, A. van Hoff, and D. Hellerstein. hjp: doc: RFC 3229: Delta encoding in HTTP, Last accessed: April 18, 2022. URL https://www.hjp.at/doc/rfc/rfc3229.html.

[81] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[82] NVIDIA. GitHub - NVIDIA/nvcomp: A library for fast lossless compression/decompression on the GPU, Last accessed: January 12, 2022. URL https://github.com/NVIDIA/nvcomp.

[83] Adnan Ozsoy and Martin Swany. CULZSS: LZSS Lossless Data Compression on CUDA. In *2011 IEEE International Conference on Cluster Computing*, pages 403–411, 2011. doi: 10.1109/CLUSTER.2011.52.

[84] Lakshmi Panneerselvam. Activation Functions | What are Activation Functions, Last accessed: April 22, 2022. URL https://www.analyticsvidhya.com/blog/2021/04/activation-functions-and-their-derivatives-a-quick-complete-guide/.

[85] Lokukaluge P Perera and Brage Mo. Ship performance and navigation data compression and communication under autoencoder system architecture. *Journal of Ocean Engineering and Science*, 3(2):133–143, 2018.

[86] Ovidiu Plugariu, Lucian Petrica, Radu Pirea, and Radu Hobincu. Hadoop ZedBoard cluster with GZIP compression FPGA acceleration. In *2019 11th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pages 1–5, 2019. doi: 10.1109/ECAI46879.2019.9042006.

[87] The Open MPI Project. Portable Hardware Locality (hwloc), Last accessed: March 30, 2022. URL https://www.open-mpi.org/projects/hwloc/.

[88] Laura Promberger, Marco Clemencic, Ben Couturier, Aritz Brosa Iartza, and Niko Neufeld. Porting the LHCb Stack from x86 (Intel) to aarch64 (ARM) and ppc64le (PowerPC). In *EPJ Web of Conferences*, volume 214, page 05016. EDP Sciences, 2019.

[89] Laura Promberger, Rainer Schwemmer, and Holger Fröning. Assessing the Overhead of Offloading Compression Tasks. In *49th International Conference on Parallel Processing - ICPP: Workshops*, ICPP Workshops '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450388689. doi: 10.1145/3409390.3409405. URL https://doi.org/10.1145/3409390.3409405.

[90] Laura Promberger, Rainer Schwemmer, and Holger Fröning. Characterization of data compression across CPU platforms and accelerators. *Concurrency and Computation: Practice and Experience*, page e6465, 2021. doi: https://doi.org/10.1002/cpe.6465. URL https://misclibrary.wiley.com/doi/abs/10.1002/cpe.6465.

[91] Laura Promberger, Holger Fröning, and Rainer Schwemmer. Characterization and Integration of Accelerated Compression for a Fair System Performance. *Submitted for Publication*, 2022.

[92] W. Qiao, J. Du, Z. Fang, M. Lo, M. F. Chang, and J. Cong. High-Throughput Lossless Compression on Tightly Coupled CPU-FPGA Platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 37–44, April 2018. doi: 10.1109/FCCM.2018.00015.

[93] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[94] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

[95] Chitta Ranjan. Build the right Autoencoder — Tune and Optimize using PCA principles. Part II | by Chitta Ranjan | Towards Data Science, Last accessed: April 25, 2022. URL https://towardsdatascience.com/build-the-right-autoencoder-tune-and-optimize-using-pca-principles-part-ii-24b9cca69bd6.

[96] Chitta Ranjan. Build the right Autoencoder — Tune and Optimize using PCA principles. Part I | by Chitta Ranjan | Towards Data Science, Last accessed: April 25, 2022. URL https://towardsdatascience.com/build-the-right-autoencoder-tune-and-optimize-using-pca-principles-part-i-1f01f821999b.

[97] Matthias Richter. Data compression in ALICE by on-line track reconstruction and space point analysis. *Journal of Physics: Conference Series*, 396(1):012043, dec 2012. doi: 10.1088/1742-6596/396/1/012043.

[98] Cliff Robinson. Microsoft Project Corsica ASIC Delivers 100Gbps Zipline Performance, Last accessed: October 25, 2019. URL https://www.servethehome.com/microsoft-project-corsica-asic-delivers-100gbps-zipline-performance/.

[99] Andrés Rodríguez, Angeles Navarro, Rafael Asenjo, Francisco Corbera, Rubén Gran, Darío Suárez, and Jose Nunez-Yanez. Exploring heterogeneous scheduling for edge computing with CPU and FPGA MPSoCs. *Journal of Systems Architecture*, 98:27 – 40, 2019. ISSN 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2019.06.006. URL http://www.sciencedirect.com/science/article/pii/S1383762119300918.

[100] RRZE-HPC. GitHub - RRZE-HPC/likwid: Performance monitoring and benchmarking suite, Last accessed: Accessed on 29th October, 2019. URL https://github.com/RRZE-HPC/likwid.

[101] Khaled Salah, K El-Badawi, and F Haidari. Performance analysis and comparison of interrupt-handling schemes in gigabit networks. *Computer Communications*, 30(17): 3425–3441, 2007.

[102] Khalid Sayood. *Introduction to Data Compression (2nd Ed.)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. ISBN 1558605584.

[103] Cedric Seger. An investigation of categorical variable encoding techniques in machine learning: binary versus one-hot and feature hashing, 2018.

[104] Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[105] Ilia Shumailov, Yiren Zhao, Robert Mullins, and Ross Anderson. To Compress Or Not To Compress: Understanding The Interactions Between Adversarial Attacks And Neural Network Compression. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 230–240, 2019. URL https://proceedings.mlsys.org/paper/2019/file/ec5decca5ed3d6b8079e2e7e7bacc9f2-Paper.pdf.

[106] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. Database Analytics Acceleration Using FPGAs. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 411–420, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1182-3. doi: 10.1145/2370816.2370874. URL http://doi.acm.org/10.1145/2370816.2370874.

[107] TensorFlow. TensorFlow, Last accessed: April 22, 2022. URL https://www.tensorflow.org/.

[108] Lucas Theis, Wenzhe Shi, Andrew Cunningham, and Ferenc Huszár. Lossy image compression with compressive autoencoders. *arXiv preprint arXiv:1703.00395*, 2017.

[109] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. Heterogeneous Resource-Elastic Scheduling for CPU+FPGA Architectures. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, HEART 2019, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450372558. doi: 10.1145/3337801.3337819. URL https://doi.org/10.1145/3337801.3337819.

[110] Peiliang Wu and William John Teahan. A new PPM variant for Chinese text compression. *Natural Language Engineering*, 14(3):417–430, 2008.

[111] Jisoo Yang, Dave B Minturn, and Frank Hady. When poll is better than interrupt. In *FAST*, volume 12, pages 3–3, 2012.

[112] Song Zebang and Kamata Sei-ichiro. Densely Connected AutoEncoders for Image Compression. ICIGP '19, pages 78–83, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450360920. doi: 10.1145/3313950.3313965. URL https://doi.org/10.1145/3313950.3313965.

[113] Max Zeyen, James Ahrens, Hans Hagen, Katrin Heitmann, and Salman Habib. Cosmological Particle Data Compression in Practice. In *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, ISAV'17, pages 12–16, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 978-1-4503-5139-3. doi: 10.1145/3144769.3144776. URL http://doi.acm.org/10.1145/3144769.3144776.

[114] Fan Zheng, Yalan Ling, Yuqing Tang, Shuai Hui, and Hongyuan Yang. A fidelity-restricted distributed principal component analysis compression algorithm for non-cable seismographs. *Journal of Applied Geophysics*, 169:29–36, 2019.

[115] Zstandard. Zstandard - Real-time data compression algorithm, Last accessed: August 2, 2019. URL https://facebook.github.io/zstd/.

[116] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 59–59, April 2006. doi: 10.1109/ICDE.2006.150.