CHALMERS
UNIVERSITY OF TECHNOLOGY

DPTC - An FPGA-Based Trace Compression

Citation for the original published paper (version of record):
Bruni, G., Johansson, H. (2020). DPTC - An FPGA-Based Trace Compression. IEEE Transactions on Circuits and Systems I: Regular Papers, 67(1): 189-197.
http://dx.doi.org/10.1109/TCSI.2019.2945179

DPTC—An FPGA-Based Trace Compression

Giovanni Bruni, *Student Member, IEEE*, and Håkan T. Johansson

***Abstract*— Recording of flash-ADC traces is challenging from both the transmission bandwidth and storage cost perspectives. This work presents a configuration-free lossless compression algorithm, which addresses both limitations, by compressing the data on-the-fly in the controlling FPGA. Thus it can easily be used directly in front-end electronics. The method first computes the differences between consecutive samples in the traces, thereby concentrating the most probable values around zero. The values are then stored as groups of four, with only the necessary least-significant bits in a variable-length code, packed in a stream of 32-bit words. To evaluate the efficiency, the storage cost of compressed traces is modeled as a baseline cost including ADC noise, and a cost for pulses that depends on amplitude and width. The free parameters and the validity of the model are determined by compressing artificial traces with varying characteristics. The compression method was also applied to actual data from different types of detectors. A typical storage cost is around 4 to 5 bits per sample. Code for the FPGA implementation in VHDL and for the CPU decompression routine in C are available as open source software, both able to operate at speeds of 400 Msamples/s.**

***Index Terms*— Analog-to-digital conversion (ADC), data acquisition, data compression, field programmable gate array (FPGA), front-end electronics, lossless compression, real-time data acquisition, open source, variable-length code, VHDL.**

I. Introduction

THIS work is motivated by developments in data handling in nuclear and particle physics. However, its applicability is not limited to those fields. Experiments in nuclear and particle physics are growing, which implies an increasing amount of data that needs to be handled. This is caused by an increase in the number of detectors employed, finer segmentation and higher event rates. Of particular interest for this work is the recording of signal traces, because this is associated with a dramatic increase of data that need to be transferred, compared with a simple digitization of pulse amplitudes.

To illustrate the development of experimental setups, we consider two front-line particle physics experiments almost 30 years apart. We compare the ATLAS (*A Toroidal LHC ApparatuS*) experiment at LHC, CERN, which started data-taking in 2009, with the UA1 (*Underground Area 1*) experiment at Sp$\bar{\text{p}}$S, CERN, which started data-taking in 1981. Concerning data production, UA1 was designed to deliver around 3 MB/s, mainly limited by the speed in writing to magnetic tape [1]. The data acquisition of ATLAS on the other hand stores around 320 MB/s [2], with much higher internal data rates. The increase of a factor 100 in recorded data rate over a time span of 30 years is compensated by the substantial improvement of commercial development in both communication and storage.

Considering the evolution of Ethernet between 1980 and 2010, we have witnessed an increase of about a factor 20 every 10 years in bandwidth [3], [4], with the major increase in the latter half of the timespan. After 2010 however, a lower rate of growth, a factor 4 every 10 years, starts to appear.

For data storage, between 1980 and 2010, the increase was on average a factor 30 every 10 years, with a peak between 1990 and 2005 where the area density doubled and prices per byte fell by half on a yearly basis [5]. Also this pace has slowed down since around 2010, with instead a factor 4 every 10 years [6], [7].

This slowdown in industry development poses new data acquisition challenges for both transmission speed and storage. A particular case when these are in high demand is when scientists are interested in storing entire traces, i.e. raw data directly from flash-ADCs, for example during testing or debugging of detectors and data processing procedures. In this case, the amount of data is much larger, easily by a factor 20–1000 [8].

One way to cope with these challenges is to increase capital expenditure to buy newer and better performing equipment. However the need to reduce costs leads to a different approach, where we aim to reduce the size of the data to be handled. This can be achieved through data compression.

A typical example of the traces considered is time-series data from flash-ADCs, which usually are slowly varying, with short intervals of larger variations due to pulses. The series data can also be information from adjacent channels, e.g. coupled strips of Si detectors, which can exhibit similar correlation characteristics.

If compression is employed as software running on a PC, only data which has already been sent from the signal acquisition unit can be reduced. This gives no reduction in the transfer rate demands. To address both limitations, an implementation of the compression directly on the FPGA, where the initial signal processing takes place, is needed. This article presents a simple yet effective lossless compression method, that can be applied to sequences of correlated data. The method allows a straightforward and fast implementation in FPGAs as well as CPUs, and is available as open source software.

This paper is structured in the following way: First, already available solutions are reviewed, followed by a description of

Manuscript received August 30, 2019; accepted September 29, 2019. Date of publication October 18, 2019; date of current version January 15, 2020. This work was supported in part by the Swedish Research Council, the Scientific Council for Natural and Engineering Sciences under Grant 2017-03839 and in part by the Council for Research Infrastructure under Grant 822-2014-6644. This article was recommended by Associate Editor M. Mozaffari Kermani. *(Corresponding author: H. T. Johansson.)*

The authors are with the Department of Physics, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (e-mail: f96hajo@chalmers.se).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSI.2019.2945179

the present routine. Optimisation possibilities, both regarding compression efficiency and resource utilisation are then discussed. This is followed by descriptions of the interfaces to the FPGA compression module and the CPU decompression code. The storage cost of both noise and pulses are then modeled, and verified using synthetic trace simulations. Finally, the achieved storage cost reduction is benchmarked using traces from actual detectors.

II. Overview of Available Solutions

Ideas for data compression on front-end electronics are not new. Scientists working on large detectors have already faced the problem of how to efficiently compress data, albeit with different boundary conditions than in our case. Both *lossy* compressions, where a part of the initial information is lost to accomplish a reduction [9], [10]; and *lossless* compressions, where the initial information can be fully reconstructed, can be achieved following different approaches. One is to discard parts of the signal with no or little information (*zero-suppression* [11]). Another approach is to use a *variable length coding* [12], such as *Huffman coding* [13] as shown in [14] or *Golomb-Rice coding*, which is used in [15]. The effectiveness of such algorithms is based on the knowledge of the probability distribution of the original data values. Usually this knowledge is gained from inspecting the whole or a representative pool of the data undergoing compression. This requires to store and to analyse a representative sample of the data during setup, in order to tune the compression configuration to the signal and ADC operation parameters. As the signal characteristics have a tendency to change within and between calibration and production data, causing operational inconveniences, such approaches are not suitable for our purpose as a generic configuration-free compression method for traces, as it causes additional work when operating detectors.

In some cases, through a *pre-processing* of the incoming data, a more advantageous probability distribution can be exploited. A common approach is the calculation of differences between values [16]–[20]. These differences may be between sampled data and a model [16] or between sampled data and a reference value (base) [17], [18] or between consecutive samples [16], [18], [19]. When dealing with signal traces, which are sampled at rates high enough that consecutive samples have values close to each other, i.e. are correlated, the latter approach delivers a distribution dominated by small values.

III. Operating Principle

The difference predicted trace compression (DPTC) presented in this paper is based on preserving only those least significant bits which hold the information necessary to recover each value. Although this does not correspond to a real Huffman coding, the result is to encode the more common smaller values, i.e. closer to zero, with shorter sequences. This approach is quite similar to the one presented in [18], where one sample works as base value and the following three samples undergo the differencing treatment. The base value can be chosen arbitrarily. We use the first value of each trace, with all following samples subject to the difference processing. The resulting differences are organised in groups of four, and all the samples in one group are stored using the same number of bits. A small header containing information about the encoding is placed at the beginning of each group.

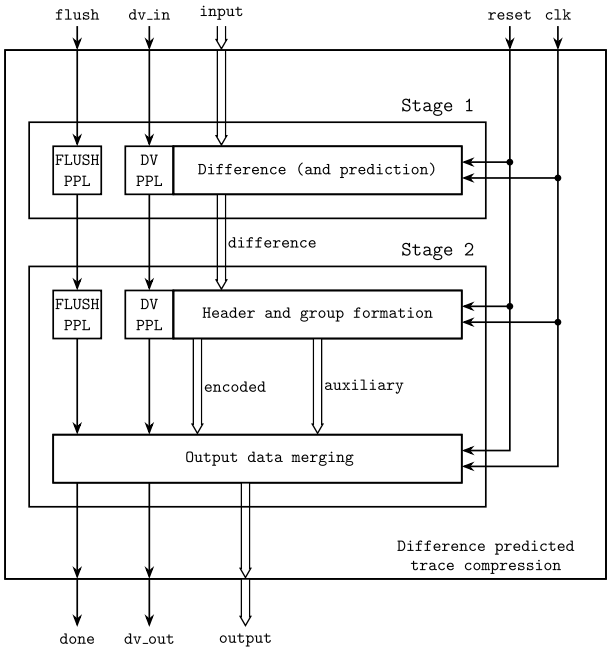


Fig. 1. Behavioural structure of the circuit implementing the DPTC algorithm. The first stage prepares the values to be encoded as differences of the input values. The second stage concerns the bit-packaging and determines the number of bits needed for each group, and prepares and shifts group headers and encoded values, merging them into the output words. One input value can be accepted each cycle, marked by the data valid (`dv`) signal. This predicate follows the data pipeline (PPL), and could in the future allow the circuit to operate even if new values are not provided every cycle. Full `output` words are signalled by `dv_out`. The end of a trace is to be marked by the `flush` signal, which, after passing the pipeline, ensures that the last data word is generated, followed by `done`.

Our implementation is organised in two steps, as shown in Fig. 1. First, the procedure calculating the differences is applied to the input data. The original samples consist of a sequence of n-bit data words, where n is given by the bit resolution of the sampling ADC. The current design allows n to have any value in the range 5–16. The second stage is responsible for packing the differences into a stream of 32-bit words.

A. Differencing Procedure

The first stage treats each value according to the following rules:

1) Calculate the difference to the previous value.
2) The later storage is due to the binary encoding slightly *asymmetric*. With a certain number of bits, it can store one more negative value than positive. With e.g. 3 bits, the eight differences $-4, -3, -2, \ldots, 3$ can be stored. For flat (noise-like) parts of a trace, any deviation from zero will generally be followed by a difference of the opposite sign. To make negative values more common than positive, a sign-changing scheme is applied: If a

Short encoding, $\Delta m = -1, 0$, or 1.

Header	Data values			
2 bits	m bits	m bits	m bits	m bits
01, 10, 11	1st diff.	2nd diff.	3rd diff.	4th diff.

Long encoding, other Δm.

Header		Data values			
2 bits	k bits	m bits	m bits	m bits	m bits
00	$\Delta m - 2$	1st diff.	2nd diff.	3rd diff.	4th diff.

Fig. 2. Layout of a group of difference values together with its header. Depending on the difference Δm in the number of bits that are needed in this group compared to the previous, the group header is characterised by a *short* (upper) or *long* (lower) encoding. The parameter k is fixed by the maximum number of bits that are needed to represent the maximum difference not covered by the short header $\max(\Delta m) = n - 3$.

stored value is negative, the next non-zero value is stored with inverted sign; while, if positive, the next is stored as is. A value of zero does not change how to store following values.

Note that in all operations, only n bits are considered, i.e. the differences are allowed to wrap (arithmetic is modulo-2^n). This does not introduce any ambiguity.

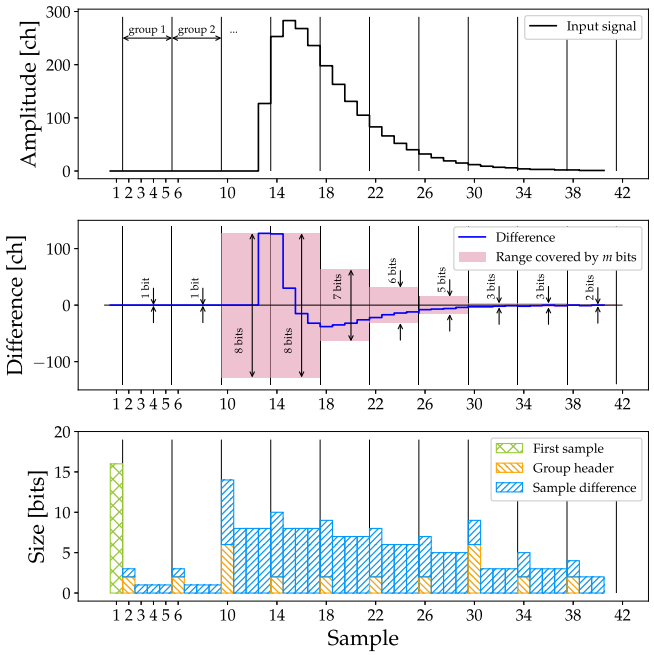


Fig. 3. Graphical representation of the differencing and group formation procedures. Shown in the upper panel is a simulated double-exponential signal (difference of double exponentials, DEXP [21]), which is used as input data, with $n = 16$. The center panel shows the differences, and how the number of bits, m, used in each group, depends on the largest difference. The lower panel shows the compressed data size in bits.

B. Group Creation

The values are stored in groups of four, using the same number of bits, m, for each value in a group. This is illustrated in Figs. 2 and 3. Since the stored values are differences, both positive and negative values must be representable (in *two's complement* representation). Since each value may require a different number of bits to be represented, the widest representation needed by any value in a group is used. The number of bits used for values in each group is stored in a group header, placed before the actual data. Considering consecutive groups, it is worth noticing that the number of bits needed will often not change much and therefore a *short* and *long* encoding of the number of bits is employed, see Fig. 2. The short header consists of two bits: if the encoded value is 1, 2, or 3, the number of bits to use for the group is the same as for the previous group with a change of -1, 0 or $+1$ bits, respectively. If the value of the two-bit short header is 0, the encoding is long and contains the full difference of bits stored per value. Since some values are already covered by the short encoding, an offset of 2 is applied to the full difference. This is encoded using k bits, which is chosen such that any needed difference, at most $n - 3$, can be stored; $k = \lceil \log_2(n-3) \rceil$, i.e. 1 bit for $n = 5$, 2 bits for $n \leq 7$, 3 bits for $n \leq 11$, and 4 bits for $n \leq 19$.

The number of bits per stored difference is interpreted with a bias of 1, meaning that storing 0 bits per value is not supported. This is a conscious choice: supporting 0-bit values (i.e. minimal encoding of groups with all value differences 0) would make the code for CPU decompression (and compression) more complicated. Since ADCs usually are operated with noise in the least significant bit, it is also expected to have limited practical use.

The data values are then stored with the necessary number of bits for the group. Each data value is stored with a bias relative to the most negative value that can be stored with the given number of bits. This simplifies decoding, as the stored value only has to be unmasked, and the bias subtracted. This avoids a cumbersome sign extension operation by the CPU decoder.

As an exception to the above rules, the first data value is stored alone and fully, using n bits. This avoids storing the entire first group of data with many bits.

C. Output Word Formation

The resulting stream of bits is then packed in 32-bit words, being filled from the least significant bits. When a value to store cannot fit, the completed output word is emitted and the remaining bits are stored in the next 32-bit output word.

Information about the number of original data values, number of data words produced by the compression and n is needed by the decompression procedure. These values are not recorded by our routine, therefore it is the responsibility of the user to retain this information.

IV. Optimisation

The algorithm described in the previous section can be optimised in different ways. However, the improvements obtained by applying additional procedures depend on many aspects, such as noise level, signal shape, and the distribution of signal

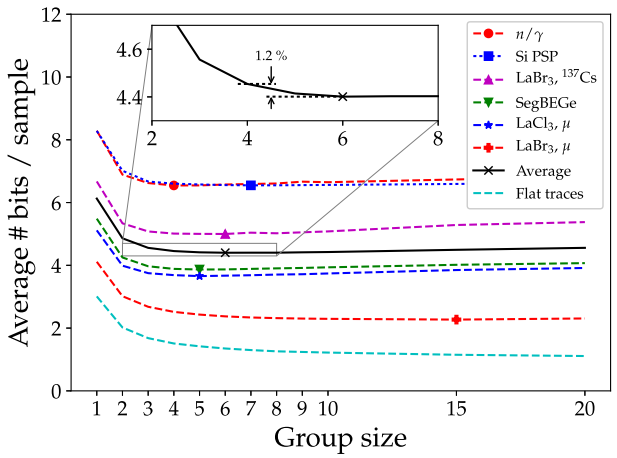


Fig. 4. Compression efficiency as a function of the group size for the actual traces presented later in Table III. The results have been averaged (geometrically) within the different sample groups, and then again to provide a total average. The minimum locations are indicated. The flat test traces are not included in the total average.

amplitudes. Note that while improving for some characteristic, an optimisation will undermine other aspects. We present a few ideas together with a short analysis of each one, discussing advantages and disadvantages.

A. Compression Factor Optimisation

1) Linear Predictor: With this additional pre-processing, the linear component of long sloped parts of a trace are removed by a second differencing of the data. This aims at a distribution of values more narrow around zero. However, for flat parts of a trace, which mainly contain noise, such a double difference leads to a wider distribution. Thus, in order to give an overall improvement, this procedure must only be applied for sufficiently long, sloped sequences. This is controlled by a heuristic using the observation that consecutive samples in unfavorable regions change sign often, or have 0 difference, and thus can be detected by a three-most-recent rule. The second differencing is switched off when at least one sign change or a zero has occurred for the previous three values.

While at first appearing to be promising for synthetic traces, from tests on actual traces, this optimisation does however not bring any improvement. This is connected to the fact that usually most of the pulses in the digitised traces only have small amplitudes, therefore the few improvements by this predictor are neutralised due to it activating spuriously in flat parts. The optimisation is implemented in the code, but deactivated by default.

2) Number of values in a group: The group size can also be varied to optimise the compression efficiency, see Fig. 4. Using smaller groups require more storage space due to the more frequent headers, while larger groups will encode unnecessary bits for more samples. The figure shows an optimum around six samples per group, with gradual losses at larger values, or steeply below three.

We have chosen to code four values in each group. The loss is about 1.2% compared to groups of six values. Fixing the number as a power of two might be useful for a future parallelized unpack code. We choose four rather than eight as this leads to shorter pipelining in the group formation part of the circuit.

TABLE I
CIRCUIT RESOURCE USAGE FOR DIFFERENT FPGA TARGETS

FPGA model	LUT occupation	FF occupation	Max clock frequency MHz
Xilinx Virtex 4	427–459	265–301	210
Xilinx Spartan 6	374–436	234–300	400
Altera Cyclone V	179–195	262–324	250
Altera Max 10	466–540	220–308	230

Table I shows the resource usage for configurations with $n = 16$, without the predictor. Note that different FPGA models have resources (e.g. look-up tables (LUT)) with different capabilities, thus the resource consumption cannot be meaningfully compared between models. The number of signal registers, flip-flops (FF), is also given. This includes $19 + 34$ FFs as inputs to, and outputs from, the actual module.

B. Circuit Optimisation

1) Additional Pipeline Stages: The achievable minimum clock cycle period in a digital circuit depends on the propagation delay of the longest combinational logic chain between register latches.

In our case the circuit is described in VHDL, where the model and grade of the FPGA that is targeted will affect which logic expression becomes the longest. Adding pipeline stages to split the longest paths helps to lower the minimum clock cycle. At the same time however, introducing a pipeline stage causes more LUTs[1] to be used, as well as flip-flops; leading to a trade-off between resource-usage and speed. In order to allow flexibility when using the code, a few generic parameters control a number of optional pipeline stages.

Since the synthesized code uses more LUTs than flip-flops, compared to the usually available ratio on FPGAs, we concentrate on the LUT usage for the circuit optimization comparisons.

By performing VHDL synthesis for all combinations of the optional pipeline stages, and directing the respective FPGA development toolchain to optimise for speed, the achievable performance as function of resource usage can be determined. The results are shown in Fig. 5 and summarized in Table I. Locations further down in the figure indicate that shorter clock periods can be used, and further to the left mean less resource consumption. For each circuit, only the results which improve the achievable clock frequency for a certain resource usage is kept, thus the short curves mainly show the improvements possible as more pipeline stages are enabled. To a smaller degree they also come from the ability of the toolchains to trade resource usage for speed. To compare with the most used constructions (adders, subtractors, comparators), 16-bit adders are also shown in the figure. The VHDL code allows the minimum period of the clock to be below 10 ns (i.e. 100 MHz)

[1]Look-up table, a basic FPGA building block. The other basic unit is signal registers, i.e. flip-flops (FF).

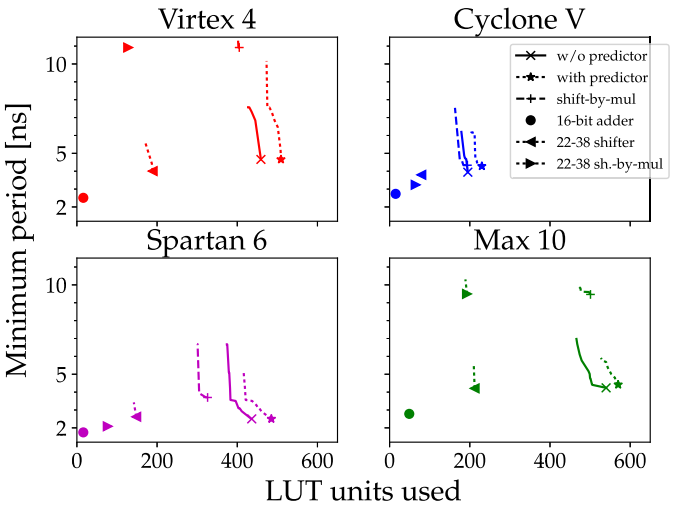


Fig. 5. Achievable circuit frequency as a function of look-up table (LUT) usage for some commercial FPGA architectures from the companies Xilinx (left column) and Altera (now Intel; right column), mainly varied by additional pipeline stages. The circuit handles one data values per clock cycle, indendent of the actual value. Three main configurations, all with $n = 16$, are evaluated: without and with the predictor, and by doing the shift using multiplier units. The resource use of a single 16-bit adder, and a 22-bit 38-position shifter are also shown. Note that different FPGA models have resources (e.g. LUTs) with different capabilities, thus the resource consumption cannot be meaningfully compared between models.

even on 10-year old FPGAs, and it can easily be configured to reach below 5 ns with additional pipeline stages. On more modern FPGAs, going below 3 ns seems rather easy. If the compression circuit is operated continuously, directly fed by the data generator (e.g. flash-ADC), the speed needs to match the sampling period, since the circuit can process one sample per clock cycle. When compressing only selected traces which first have been recorded into temporary memory buffers, a slower clock can be used for the compression circuit.

The single most expensive component of the circuit is the barrel shifter, which aligns the encoded data at the next position in the output word. For $n = 16$, the shifter input is 22 bits wide, with the additional 6 bits coming from the potentially long encoded header. The shift amount is in the range 0 to 37, inclusive. 0 to 31 depends on how many bits already are used in the output word. The additional 0, 4, or 6 positions depend on the header (long, short, or none). This gives a 60 bit output. The cost and performance of the shifter units are also shown in Fig. 5.

2) Barrel Shifter vs. Multiplier Units: A barrel shifter on FPGAs is normally realised as one multiplexer for each output bit (sharing some parts of the first stage selectors of each multiplexer). Since it also can be expressed as a multiplication of the input value with 2^i, where i is the shift value, it can also be implemented using multiplier units in FPGAs. For the second factor, the input value is generated as 2^i, i.e. a one-hot encoding of the shift amount.

One could imagine this to be beneficial when generic LUT resources are scarce, however for the cases tested, it is not. The generation of the 2^i input value is rather expensive, as it requires 2^i individual selectors. Also the combination of the output values from the several multiplier units, often 9×9 or 18×18 wide, is rather expensive.

The resource usage for 22-bit, 38-position left-shifters implemented in the two ways are also compared in Fig. 5.

The results in Fig. 5 and Table I are for $n = 16$. Similar tests for $5 \leq n < 16$ give that for each bit removed, the needed number of LUTs shrinks on average by 4–5%, and the attainable minimum period required decreases by 1–2%, depending on FPGA model.

V. VHDL Module Interface

The interface to the VHDL compression module is a single entity, with input and output signals as seen at the top and bottom of Fig. 1. Optional pipeline stages are configured using a generic map.

The circuit inputs are:

- `clk`: clock signal;
- `reset`: reset signal, given for at least as many cycles as the pipeline has stages;
- `input`: n-bit data value to compress;
- `dv_in`: data valid signal: set to '1' every clock cycle an input value is provided;
- `flush`: flush signal: set to '1', after the last input value has been given. Held until `done` reported back. This forces the last output word to be emitted, especially when it is not fully occupied.

The output signals are:

- `output`: 32-bit output data word;
- `dv_out`: data valid signal: '1' every time the output word is filled, signaling the presence of a completed data word to be stored;
- `done`: informs that the last input value has been processed and the final output word was produced (possibly in the current cycle).

VI. Decompression

The decompression is performed by one C function with the following parameters:

- `compr`: pointer to the 32-bit words of the compressed input buffer;
- `ncompr`: number of elements in the input buffer;
- `output`: pointer to a buffer of 16-bit items for the decompressed values;
- `ndata`: number of original/decompressed values;
- `bits`: number of bits of each value that was stored (n). This must be the same as the number configured during compression.

On success, 0 is returned, otherwise a non-zero value.

The routine will report decompression failure on malformed compressed data, e.g. if there are non-zero bits left in the input buffer, or when entire words have not been used. The decompression routine will not read items beyond the end of the source buffer even if it runs out of data, e.g. due to a corrupted data stream. Table II shows the typical performance, which only has a small dependence on the actual data values.

TABLE II
PERFORMANCE OF THE DECOMPRESSION ROUTINE ON VARIOUS CPUS

CPU Model	Speed (GHz)	Released	Time/sample (ns)
Xeon E3-1285v6	4.5	2017	2.0 – 2.2
Xeon E3-1276v3	4.0	2014	2.5 – 3.1
Xeon X5450	3.0	2007	5.3 – 6.2
PPC 7455	1.0	2002	28 – 36

Table II shows the single-threaded decompression times per sample for the actual traces presented later in Table III. With 16-bit data samples, and a decompression time on modern hardware smaller than 2.5 ns/sample, the decompressed rate is larger than 800 MB/s, i.e. well comparable to solid state drives (SSDs).

VII. COMPRESSION EFFICIENCY—STORAGE COST

The contributions to the compressed data size can be divided in two parts:

1) The cost of storing traces with no pulses, i.e. only containing the digitization noise. This is described as a cost per sample.
2) The cost of storing a pulse, described as an additional cost for the entire pulse.

There is a natural interplay between the two, as the noise affects the additional cost to store a pulse. This effect is also addressed below.

In the following, we use the variables c for cost and b for bits. To specify these, subscripts are used: $\mathcal{N}$ for noise, $\mathcal{T}$ for trace, $\mathcal{S}$ for sample, $\mathcal{P}$ for pulse, and $\mathcal{B}$ for a small pulse (bump). Gaussian noise is described by its amplitude $\sigma_\mathcal{N}$. The amplitude and width (std. dev.) of Gaussian-shaped pulses are given by $A_\mathcal{P}$ and $w_\mathcal{P}$.

A. Bare Trace Cost

The cost of storing a trace without pulses has two parts: the size of the headers and the size of the encoded values, i.e. the differences.

The cost of storing the differences depends on the noise content, most easily expressed as the number of bits of noise $b_\mathcal{N} = \log_2 \sigma_\mathcal{N}$.

Ignoring the pecularities of the first group, which may require a long header encoding, the estimated cost for a trace $\langle c_\mathcal{T} \rangle$ will be proportional to its length $n_\mathcal{T}$:

$$\langle c_\mathcal{T} \rangle = n + (n_\mathcal{T} - 1)\langle c_\mathcal{S} \rangle + 15.5. \tag{1}$$

The first sample has a fixed cost n. The constant 15.5 accounts for the average number of unused bits in the last output word at the end of a trace. A first approximation, denoted by the tilde, for the average cost per noise sample is

$$\langle \widetilde{c_\mathcal{S}} \rangle = 0.5 + b_\mathcal{N} + 1 + o. \tag{2}$$

The first half bit comes from the short group header, using two bits every four samples. The additional one comes from differences encoding both positive and negative entries, i.e. effectively a sign bit. The term o is an overhead, since the grouping of values causes some more bits than necessary to be used. To model the transition from very small noise levels, where the total cost is 1.5 bits/sample, to the proportional regime, a smooth transition function $g(x) = \frac{1}{f}\log_2(1 + x^f)$ is used for $b_\mathcal{N}$, with $x = \sigma_\mathcal{N}$. As wanted, $g(x) \to 0$ as $x \to 0$ and $g(x) \to \log_2 x$ for $x \gg 1$, while the parameter f controls the smoothing. This yields:

$$\langle c_\mathcal{S} \rangle = 0.5 + \frac{1}{f}\log_2(1 + {\sigma_\mathcal{N}}^f) + 1 + o. \tag{3}$$

This is illustrated for Gaussian and uniform noise in Fig. 6, where good fits are achieved with $f = 8$. For uniform noise, the range of differences is twice as large as the value distribution (due to also encoding negative entries), explaining the use of, on average, one more bit per sample in addition to the short header and $b_\mathcal{N}$. This is modeled by (3) shown as a solid line. For Gaussian noise, the distribution of differences between consecutive samples is wider by a factor $\sim \sqrt{2}$ than the original distribition, and any large value in a group of four leads to longer encodings. The further small fractional costs per sample are in both cases likely given by the occasional use of long group headers.

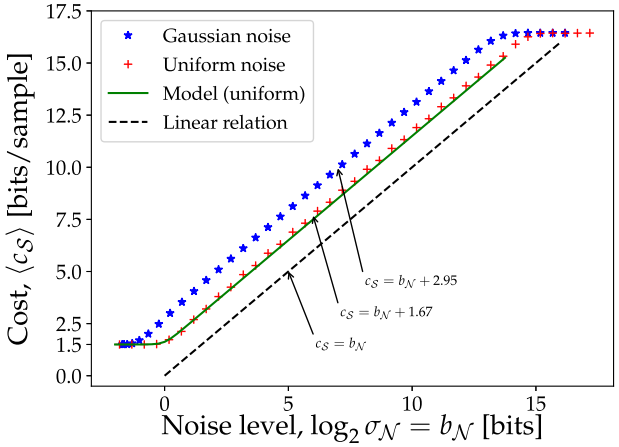


Fig. 6. Average cost per sample depending on the number of noisy bits. The minimum cost per sample is 1.5 bits, given by one bit per sample and a short header of two bits every group of four samples. The model is compared to actual compression of traces with either Gaussian or uniform noise. For Gaussian noise the standard deviation of the distribution expresses the number of noisy bits. For uniform noise instead the number of noisy bits represents the span of the random value distribution.

B. Pulse Cost

The cost of a pulse is best described as the total cost of the pulse, and not a cost in bits per sample. For the following discussion, pulses are assumed to have a Gaussian shape, as opposed to the double exponential function considered in Fig. 3. Detector pulses can be considered as composed of two parts with different time constants (i.e. widths) for the rising and falling parts. Even if this may be a rather rough approximation of real pulses, especially for the trailing part, it is practical, since Gaussian functions are efficiently and familiarly described using their widths and amplitudes.

Since it is differences that are stored, the important parameter is not the amplitude $A_\mathcal{P}$ of a pulse, but its steepest slope, which scales as $\frac{A_\mathcal{P}}{w_\mathcal{P}}$. As a first approximation, denoted by the tilde, the cost is proportional to the number of bits needed to store these differences, as well as the width of the pulse,

$$\langle \widetilde{c_\mathcal{P}} \rangle = a w_\mathcal{P} \log_2 \left(\frac{A_\mathcal{P}}{w_\mathcal{P}} \right). \tag{4}$$

The scale is given by the proportionality constant a. It turns out that this formula works rather well, if modified to account for the facts that even for small pulses, costs are not negative (by adding 1 inside the logarithm and control parameter b), and that very narrow pulses still will affect the storage size of at least one entire group (d within the square root):

$$\langle c_{\mathcal{P}} \rangle = a\sqrt{w_{\mathcal{P}}^2 + d^2}\,\frac{1}{b}\log_2\left(1 + \left(\frac{A_{\mathcal{P}}}{w_{\mathcal{P}}}\right)^b\right). \tag{5}$$

The modification is thus adjusted by the control parameters b and d.

C. Pulse-Noise Interaction

The above description (5) works in the limit where the pulse is large compared to the background noise. When this is not the case, the additional cost of storing the pulse will be *smaller*, since the pulse-associated part of the differences to some extent will be covered by the noise storage cost. This can be modeled by

$$\langle c_{\mathcal{B}} \rangle = \sqrt{\langle c_{\mathcal{P}} \rangle^2 + \langle c_{\mathcal{N}\mathcal{B}} \rangle^2} - \langle c_{\mathcal{N}\mathcal{B}} \rangle. \tag{6}$$

The correction is the cost of storing the noise for a stretch of samples proportional to the pulse width:

$$\langle c_{\mathcal{N}\mathcal{B}} \rangle = q\sqrt{w_{\mathcal{P}}^2 + d^2}\,\frac{1}{f}\log_2\left(1 + {\sigma_{\mathcal{N}}}^f\right). \tag{7}$$

q is a proportionality constant.

D. Storage Cost Verification—Synthetic Traces

The storage cost described above and culminating in (6) has been verified by simulating a large number of traces with Gaussian pulses, where the parameters $A_{\mathcal{P}}$, $w_{\mathcal{P}}$, and $\sigma_{\mathcal{N}}$ were varied. A global fit suggests the following values for the control parameters: $a = 5.6$, $b = 1.3$, $q = 33$, $d = 2.6$ and $f = 7.7$, with a parameter uncertainty of up to 10%. Fig. 7 shows the $\sigma_{\mathcal{N}} = 2.0$ case.

Simulations were performed by building, for each set of parameters, a set of 15×10^6 traces, each made of 500 samples, with Gaussian noise $\sigma_{\mathcal{N}}$. In each, a Gaussian pulse ($A_{\mathcal{P}}$, $w_{\mathcal{P}}$) was added to the trace. To average over discretisation effects, both the (noise) baseline and the center of the pulse were randomised, trace by trace, with fractional offsets.

Although Fig. 7 shows a good agreement between Equation (6) and the data, larger differences emerge for small values of $A_{\mathcal{P}}$ and $w_{\mathcal{P}}$. These correspond to the limits handled by the modifications between (4) and (5), which are thus seen to only partly address these edge effects.

E. Storage Cost Verification—Actual Traces

Table III shows the compression efficiencies for some different collections of actual data. They are compared to the common gzip [22] and xz [23] generic compression routines (at their normal setting). For the generic routines, all data of each file was stored in a binary file with 16-bit values. For a fair comparison, the overhead size of storing an empty compressed file was subtracted. In general, the DPTC results

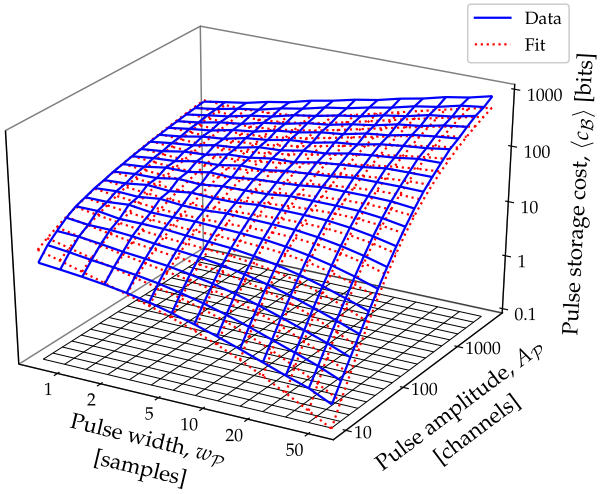


Fig. 7. Cost to store a Gaussian pulse as a function of the pulse amplitude and width. This is shown for both compressed simulated data (solid lines) and the model (6) (dotted lines). The input traces are built adding Gaussian noise, in this case with a sigma of 2.0, to the Gaussian pulse. The cost of storing only the Gaussian noise is then subtracted from the cost of storing the trace, leaving the cost of storing the pulse. The fit is done using model (6) on the total amount of data and minimizing relative differences.

are quite similar to the LZMA results, and well below the gzip results. The main exception are the $LaBr_3$ collections marked [a], where the data is very flat (virtually no noise) except for the pulses. Here the DPTC routine still uses its minimum of at least 1.5 bits/sample. This effect is also seen for the three synthetic traces marked [b], which have constant values.

Since Huffman encoding [13] is a common approach for compression where the typical distribution of values is known, the actual traces have for comparison purposes also been compressed using this approach. It is applied after a difference stage, with the Huffman encodings individually optimised for each data set. To allow average costs below one bit per sample for very flat traces, encodings of up to four consecutive values using one symbol were also allowed, when such stretches of values would account for more than 1% of the symbols. In these tests, the 1% threshold was only passed for the cases marked [a] and [b]. Overall, the Huffman compression scheme delivers results slightly better than both the DPTC routine and the generic compression routines, but needs to be optimised to the characteristics of the signals.

Finally, note how close the costs per sample are to the expectations for only storing the respective noise content, showing that the storage cost contributions from pulses are negligible.

F. Caveat Emptor—How to Ignore ADC Noise

In case the original data contains an excessive number of least-significant bits with noise that shall not be stored, they must be shifted out of the original data before the values are given to the DPTC compression routine. Just masking them out will *not* improve the compression efficiency, as the routine is looking for the most significant bit of the differences that need to be stored. On the other hand, using a compressor with n larger than necessary causes little extra cost. Few, if any, extra

TABLE III

COMPRESSION EFFICIENCY OF THE DPTC ALGORITHM FOR ACTUAL TRACES, CATEGORISED BY DETECTOR TYPE AND DETECTED RADIATION, AND COMPARED TO POPULAR GENERAL-PURPOSE COMPRESSION METHODS AND HUFFMAN ENCODING

Label	Category	Details	Traces #	Samples #	$\langle A_{\mathcal{P}} \rangle_{\mathrm{g.}}$	$\sigma_{\mathcal{N}}$	$\langle c_{\mathcal{S}} \rangle$	DPTC	gzip	xz(LZMA)	Huff.
								— — — — Bits/sample — — — —			
a	γ in segmented BEGe	core signal	40	5000	78.3	2.16	4.06	**3.89**	5.54	4.04	3.58
b		segment 1	40	5000	27.3	2.16	4.06	**3.86**	4.87	3.89	3.55
c		segment 5	40	5000	53.2	2.21	4.09	**3.91**	5.54	4.13	3.59
d	n/γ discrimination	Ionisation chamber	200	200	907	71.2	9.10	**9.16**	11.37	9.78	8.75
e		n-det. anode	200	200	226	4.88	5.24	**5.36**	6.63	5.32	5.12
f		n-det. cathode	200	200	220	6.20	5.58	**5.71**	7.00	5.62	5.46
g	position-sensitive Si pin-diode	α-particles	50	1000	852	29.7	7.84	**7.81**	11.07	8.10	7.38
h		^{40}Ar	50	1000	638	6.36	5.62	**5.58**	9.37	6.23	5.24
i	γ from ^{137}Cs in $LaBr_3$	no signal split	100	200	534	5.30	5.36	**5.55**	7.91	6.33	5.47
j		signal split 1:2	100	200	292	3.90	4.91	**5.08**	7.18	5.67	4.98
k		signal split 1:4	100	200	194	3.23	4.64	**4.81**	6.69	5.24	4.68
l		signal split 1:8	100	200	122	3.05	4.56	**4.65**	6.37	5.06	4.43
m	cosmic μ in $LaBr_3$, varying HV of PMT	350V [a]	100	600	9.2	0.25	0.94	**1.67**	0.65	0.49	0.65
n		400V [a]	100	600	19.4	0.25	0.94	**1.67**	0.84	0.63	0.78
o		450V	100	200	921	4.28	5.05	**5.55**	8.36	6.42	5.76
p	cosmic μ in $LaCl_3$, different digitizers	CAEN DT5730	100	400	301	3.88	4.90	**5.00**	7.23	5.47	4.89
q		CAEN DT5751	100	400	40.6	0.86	2.73	**2.72**	3.94	2.82	2.64
r	Flat traces [b]	all values 0	1	1000	0	0	-	**1.51**	0.28	0.67	0.26
s		all values 10	1	1000	0	0	-	**1.51**	0.28	0.67	0.26
t		all values 100	1	1000	0	0	-	**1.51**	0.28	0.67	0.26

In Table III, the noise content of each trace collection is characterised by $\sigma_{\mathcal{N}}$, which is calculated from the distribution of the differences between each sample, and the average of its four closest neighbours on each side. The pulse amplitudes $A_{\mathcal{P}}$ are represented as a geometric average of the difference between the largest and smallest sample value in each trace. This slightly overestimates the pulse amplitudes, due to the noise broadening, but since $A_{\mathcal{P}} \gg \sigma_{\mathcal{N}}$, it is still clear that the traces contain pulses. The expected costs for storing the noise, as suggested by Fig. 6, are calculated as $\langle c_{\mathcal{S}} \rangle = \log_2 \sigma_{\mathcal{N}} + 2.95$. The $LaBr_3$ collections marked [a] are very flat (virtually no noise) except for the pulses, causing the DPTC costs to be dominated by its minimum of at least 1.5 bits/sample. This also applies to the synthetic constant-value traces marked [b].

bits will be used; since mainly k will potentially be affected, see Fig. 2.

Note that the choice of omitting least-significant bits is a delicate decision. The finally achievable resolution of a measurement may be improved by retaining some additional least-significant bits, since it may allow analysis of the later de-compressed traces to partially recover the effects of quantization error and differential non-linearity in the ADC, by averaging or fitting.

When applicable, in oversampled parts of a trace, much larger savings than obtained through omitting some least-significant bit may be obtained through downsampling the information by summing adjacent samples before compression, thus storing fewer samples, but with better resolution.

VIII. CONCLUSION

A lossless compression routine which addresses both the transmission bandwidth and storage cost challenges associated with recording flash-ADC traces has been presented. The routine can be directly integrated in front-end electronics and can handle data streams on-the-fly at rates of 400 Msamples/s in the controlling FPGA. Calculation of the differences between consecutive trace samples concentrated the most frequently occuring values around zero. The compression was concluded by storing the values in groups of four, yielding a simple yet effective variable-length code, by only storing the necessary least-significant bits, in a stream of 32-bit words.

A model for the storage cost was developed, by first considering the influence of the group headers as well as the retained ADC noise. The additional cost of storing a pulse was expressed in terms of its amplitude and width. By compressing a large set of artificial traces with varying characteristics, both the free parameters and the validity of the model were determined.

The method was then applied to actual data from different kinds of detectors. The compression efficiency was found to be comparable to popular general-purpose compression methods (gzip and xz). It was shown that the dominating cost of storing actual traces is generally given by the retained ADC noise, and not the pulses. It is therefore important for users to carefully assess how many least-significant bits shall be kept, in case they are noisy. Except for that, there are no parameters that need to be adapted, which is of particular interest for experiments employing hundreds or thousands of detector channels.

Computer code for the FPGA implementation in VHDL and for the CPU decompression routine in C are available for download [24] as open source software.

ACKNOWLEDGMENT

The authors would like to extend their thanks to O. Schulz, B. Löher, S. Storck, and P. Díaz Fernández for providing test data, and to A. Heinz and D. Radford for valuable discussions.

REFERENCES

[1] A. Astbury *et al.*, "The UA1 calorimeter trigger," *Nucl. Instrum. Methods Phys. Res. A, Accel. Spectrom. Detect. Assoc. Equip.*, vol. 238, nos. 2–3, pp. 288–306, 1985.

[2] ATLAS Collaboration. (Jun. 2012). *ATLAS Fact Sheet*. Accessed: Dec. 5, 2018. [Online]. Available: http://cds.cern.ch/record/1457044/files/ATLAS fact sheet.pdf

[3] V. S. Latha and D. S. B. Rao, "The evolution of the Ethernet: Various fields of applications," in *Proc. Online Int. Conf. Green Eng. Technol. (IC-GET)*, Nov. 2015, pp. 1–7.

[4] Ethernet Alliance. (Feb. 2018). *2018 Roadmap Graphics*. Accessed: Dec. 6, 2018. [Online]. Available: https://ethernetalliance.org/the-2018-ethernet-roadmap/

[5] R. J. T. Morris and B. J. Truskowski, "The evolution of storage systems," *IBM Syst. J.*, vol. 42, no. 2, pp. 205–217, 2003.

[6] R. Wood, "Future hard disk drive systems," *J. Magn. Magn. Mater.*, vol. 321, no. 6, pp. 555–561, Mar. 2009.

[7] A. Nordrum, "The fight for the future of the disk drive," *IEEE Spectr.*, vol. 56, no. 1, pp. 44–47, Jan. 2019.

[8] S. Paschalis *et al.*, "The performance of the gamma-ray energy tracking in-beam nuclear array GRETINA," *Nucl. Instrum. Methods Phys. Res. A, Accel. Spectrom. Detect. Assoc. Equip.*, vol. 709, pp. 44–55, May 2013.

[9] D. Falchieri, E. Gandolfi, and M. Masotti, "Evaluation of a wavelet-based compression algorithm applied to the silicon drift detectors data of the ALICE experiment at CERN," *Nucl. Instrum. Methods Phys. Res. A, Accel. Spectrom. Detect. Assoc. Equip.*, vol. 527, no. 3, pp. 580–590, 2004.

[10] A. Nicolaucig, M. Ivanov, and M. Mattavelli, "Lossy compression of TPC data and trajectory tracking efficiency for the ALICE experiment," *Nucl. Instrum. Methods Phys. Res. A, Accel. Spectrom. Detect. Assoc. Equip.*, vol. 500, nos. 1–3, pp. 412–420, 2003.

[11] A. Werbrouck *et al.*, "Image compression for the silicon drift detectors in the ALICE experiment," *Nucl. Instrum. Methods Phys. Res. A, Accel. Spectrom. Detect. Assoc. Equip.*, vol. 471, no. 1, pp. 281–284, 2001.

[12] K. Sayood, *Introduction to Data Compression*, K. Sayood, Ed., 4th ed. San Mateo, CA, USA: Morgan Kaufmann, 2012.

[13] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. Inst. Radio Eng.*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.

[14] G. Mazza and S. Cometti, "The front-end data conversion and readout electronics for the CMS ECAL upgrade," *J. Instrum.*, vol. 13, no. 3, 2018, Art. no. C03003.

[15] R. Ammendola *et al.*, "Natrium: Use of FPGA embedded processors for real-time data compression," *J. Instrum.*, vol. 6, no. 12, 2011, Art. no. C12036.

[16] C. Patauner, A. Marchioro, S. Bonacini, A. U. Rehman, and W. Pribyl, "A lossless data compression system for a real-time application in HEP data acquisition," *IEEE Trans. Nucl. Sci.*, vol. 58, no. 4, pp. 1738–1744, Aug. 2011.

[17] J. Duda and G. Korcyl, "Designing dedicated data compression for physics experiments within FPGA already used for data acquisition," 2015, *arXiv:1511.00856*. [Online]. Available: https://arxiv.org/abs/1511.00856

[18] R. Kobayashi and K. Kise, "A high performance FPGA-based sorting accelerator with a data compression mechanism," *IEICE Trans. Inf. Syst.*, vol. E100-D, no. 5, pp. 1003–1015, 2017.

[19] J. Badier, P. Busson, A. Karar, D. Kim, G. Kim, and S. Lee, "Reduction of ECAL data volume using lossless data compression techniques," *Nucl. Instrum. Methods Phys. Res. A, Accel. Spectrom. Detect. Assoc. Equip.*, vol. 463, nos. 1–2, pp. 361–374, 2001.

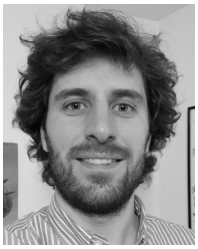
[20] A. Biasizzo and F. Novak, "Hardware accelerated compression of LIDAR data using FPGA devices," *Sensors*, vol. 13, no. 5, pp. 6405–6422, May 2013.

[21] G. Wu, "Shape properties of pulses described by double exponential function and its modified forms," *IEEE Trans. Electromagn. Compat.*, vol. 56, no. 4, pp. 923–931, Aug. 2014.

[22] *GNU Gzip*. Accessed: Feb. 12, 2019. [Online]. Available: https://www.gnu.org/software/gzip/

[23] *XZ Utils*. Accessed: Feb. 12, 2019. [Online]. Available: https://tukaani.org/xz/

[24] *DPTC—Firmware for FPGA Trace Compression*. Accessed: Mar. 26, 2019. [Online]. Available: http://fy.chalmers.se/subatom/dptc/



Giovanni Bruni (S'11) received the M.S. degree in electronic engineering from the Università degli Studi di Padova, Padova, Italy, in 2014 and the M.S. degree in physics from the Chalmers University of Technology, Göteborg, Sweden, in 2018, where he is currently pursuing the Ph.D. degree in physics.

His current research interests include detector development, data acquisition systems, radiation effects on digital circuits, FPGA development, and data analysis of nuclear physics experiments.

Håkan T. Johansson was born in Halmstad, Sweden, in 1977. He received the M.S. degree in engineering physics and the Ph.D. degree in physics from the Chalmers University of Technology, Göteborg, Sweden, in 2002 and 2011, respectively.

In 2001, he was a Summer Student with CERN, and from 2003 to 2007, a Research Assistant with GSI, Darmstadt, Germany. Since 2010, he has been a Research Engineer with the Subatomic Physics Group, Physics Department, Chalmers University of Technology. His research interests include the efficient use of computers in all aspects of experiment preparations, execution, and analysis and high-performance computations for theory.

Dr. Johansson is a member of the Swedish Nuclear Physics Society.