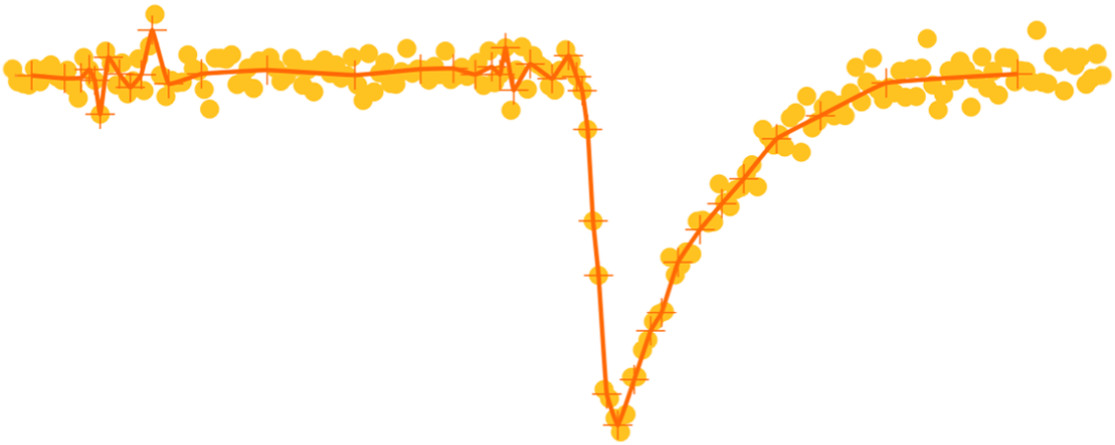

Adaptive downsampling of traces with FPGAs

Master's thesis in Complex adaptive systems

ANTON FREDRIKSSON

LUKAS RAHMN

Department of physics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Master's thesis 2020

Adaptive downsampling of traces with FPGAs

ANTON FREDRIKSSON
LUKAS RAHMN

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of physics
Division of Subatomic, high energy and plasma physics
Subatomic Physics
Chalmers University of Technology
Gothenburg, Sweden 2020

Adaptive downsampling of traces with FPGAs

ANTON FREDRIKSSON
LUKAS RAHMN

Supervisor: Håkan Johansson, Department of physics
Examiner: Andreas Heinz, Department of physics

Master's Thesis 2020
Department of physics
Division of Subatomic, high energy and plasma physics
Subatomic Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An example trace (yellow dots) together with the resulting trace (orange crosses) after adaptive downsampling.

Adaptive downsampling of traces with FPGAs
Anton Fredriksson, Lukas Rahmn
Department of Physics
Chalmers University of Technology

Abstract

Modern nuclear physics experiments produce large amounts of data. It has become custom to record and store signal traces, which can benefit from compression algorithms. General-purpose compression methods do not exploit the characteristics of the traces and are ill-suited for implementation on Field-Programmable Gate Arrays (FPGAs). We propose a lossy algorithm based on downsampling. Data reduction is achieved by replacing variable-length groups of samples with their average. The lengths of the groups are chosen according to their significance, which is determined based on the estimated signal noise. This results in an adaptive downsampling of the signal trace. The compression routine also assists subsequent analysis, by reducing the stored signal fluctuations due to noise. A detailed algorithm description is provided, accompanied by openly available implementations for both FPGAs and PCs (VHDL and C++).

Keywords: VHDL, Data acquisition, Front-end electronics, Downsampling, Lossy compression.

Acknowledgements

We would like to thank Håkan Johansson and Andreas Heinz, our supervisor and examiner respectively, for all the help, time and interesting (at times surprisingly extensive) conversations afforded to us. Not only did you aid us in producing an interesting thesis, but made the journey there fun. Thank you!

Anton Fredriksson and Lukas Rahmn, Gothenburg, May 2020

Glossary

ADC	Analog-to-Digital Converter
ADS	Adaptive DownSampling
DWT	Discrete Wavelet Transform
FPGA	Field-Programmable Gate Array
FIFO	First-In First-Out
IIR	Infinite Impulse Respone
LUT	LookUp table
MAD	Mean Absolute Deviation
RAM	Random access memory
STD	Standard Deviation
SoC	System-on-a-Chip

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Aim . . . 2
- 1.3 Scope . . . 3
- 1.4 Previous work . . . 4
- 1.5 Thesis outline . . . 5

2 Algorithm description **7**
- 2.1 Principle of operation . . . 9
- 2.2 Compressor design . . . 12
  - 2.2.1 Ratio selection . . . 14
  - 2.2.2 Bit encoding and compression ratio . . . 16
- 2.3 Standard deviation estimation of the noise . . . 16
  - 2.3.1 Relation to conventional STD . . . 17
- 2.4 Decompression . . . 18
- 2.5 Boundary conditions . . . 19
- 2.6 DPTC Integration . . . 19

3 Software development **23**
- 3.1 Matlab implementation . . . 24
- 3.2 C++ implementation . . . 24
  - 3.2.1 Website . . . 24
- 3.3 VHDL . . . 25

4 Results **27**
- 4.1 Compression performance . . . 27
- 4.2 Noise STD estimator characteristics . . . 29
- 4.3 FPGA resource utilization and timing . . . 32

5 Discussion **35**

Bibliography **39**

1

Introduction

1.1 Background

This project concerns software development in the context of large-scale nuclear physics experiments. The experimental setup uses a heavy ion accelerator to produce beams of radioactive ions at relativistic energies. Those interact with thin targets with less than 10% reaction probability. The reaction products are detected with a multitude of detectors placed at key positions before and after the target. The detector front-end electronics employs a large number of field programmable gate arrays (FPGA) for control and signal processing.

The use of FPGAs provides flexibility by allowing changes of the front-end operation whilst being able to meet the imposed timing and performance requirements. The FPGAs employed at detector front-ends must process the sampled detector signals in real time while remaining synchronized with the whole setup. The amount of data produced by these front-ends can be large, especially if the complete time-series is of interest, which then require vast amounts of storage. Handling and storage can be both cumbersome and/or expensive, thus one does not want to record more than needed. Large data streams also imply a large bandwidth requirement for the transfer links employed between front end electronics and storage facilities. If one can somehow reduce the amount of the data produced, then either one lessens the infrastructure requirements, or for the same equipment increases the amount of events that can be processed.

The data of concern here are signal traces, which are short waveforms of recorded detector data. Usually a trace is produced by an Analog to Digital Converter (ADC) which samples and digitizes an analogue and continuous voltage many times a second. The digitized voltage values are called samples and are stored as fixed size integers. The size in bits of these integers depends on the precision of the used ADC, where common sizes are ten to sixteen bits. Thus, a trace is an array of such integer samples, an example trace is shown in figure 1.1. The bits needed to represent a trace is the number of bits per sample times the length of the trace, i.e the total number of samples.

This thesis will address how such a trace can be compressed, i.e reduced in size,

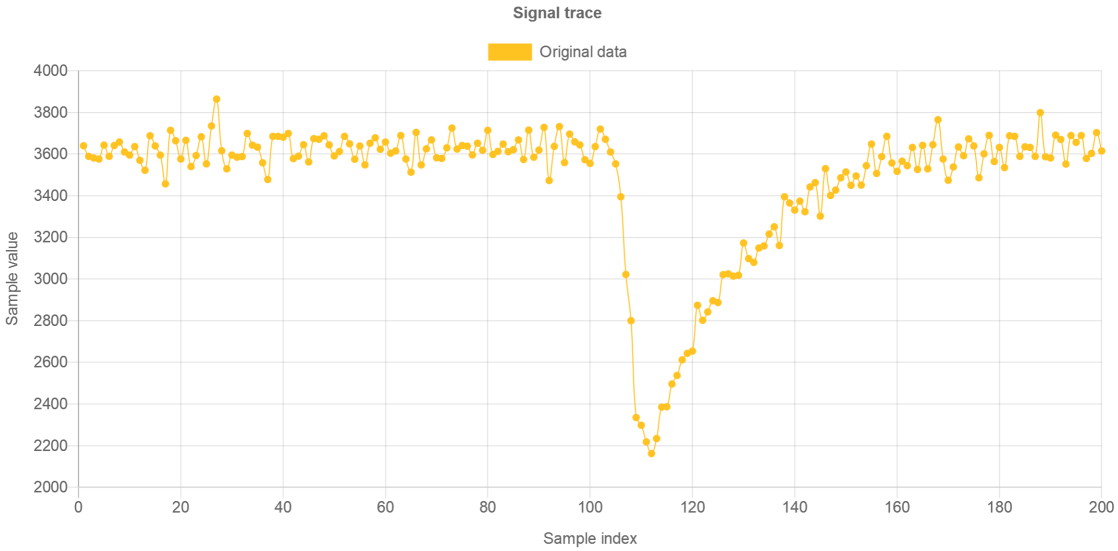

Figure 1.1: An example of a detector signal trace. Each yellow dot represents a signal sample, i.e a data point. The line between samples is added to guide the eye, it is not indicative of how the original voltage signal progressed between two samples. Such information is lost when the continuous signal is sampled.

by careful selection of what data to store and an efficient encoding of the selection. By testing signal significance compared to noise, and selective averaging of sample groups, the stored signal noise is reduced thus increasing the compression ratio. Using our algorithm, which will be presented in chapter 2, real world traces are recorded with an average of one to three bits per sample, depending on settings, instead of the 10 to 14 bits/sample of the original traces. For the trace in figure 1.1, this average is 2.77 bits per sample, the reconstruction of this compressed trace is shown in figure 1.2. To experiment oneself with the compression system please consult our webpage `http://fy.chalmers.se/subatom/ads/compressor/`. A description of the webpage is provided in section 3.2.1.

1.2 Aim

The contribution of this project is the development of a data compression scheme for use in the front-end FPGAs. We produce a reference implementation, consisting of both an FPGA-based compressor and a software implementation of both the compressor and decompressor, to facilitate quick and easy integration of compression in pre-existing front-end FPGAs. The reference implementation has been thoroughly tested for defects to ensure proper operation.

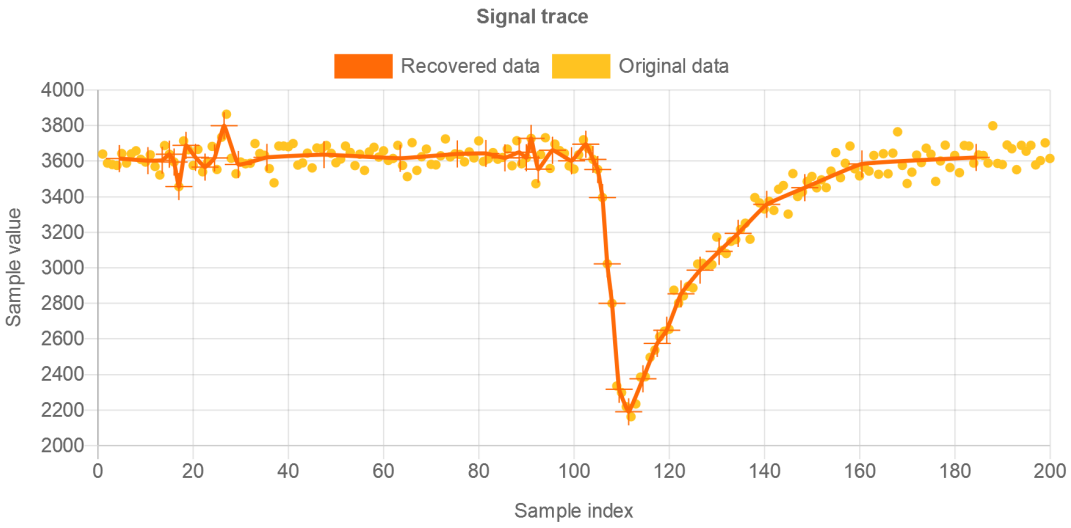

Figure 1.2: A example trace before (yellow) and after compression (orange). The average number of bits per samples needed to store the compressed trace is 2.77 whilst the original trace was recorded with 12 bits per sample. Once again the orange line is just to aid the eye, no interpolation is done by the compression routine. It is up to the user of the system to reconstruct the trace from the provided data points (the orange crosses). Trigger level is $K = 3$, see chapter 2 for parameter description.

1.3 Scope

The problem of signal compression is widely researched, and numerous methods have been developed. Examples span from classic methods such as Huffman-coding [1, 2, 3], Arithmetic-coding and run-length coding to the less intuitive transform-based compression algorithms such as the discrete cosine transform employed for the JPEG standard [3]. Compression algorithms can be divided into two categories. Lossless schemes exploit redundancies, patterns and shared knowledge. They allow for the creation of alternate representations of the data, from which the original data can be reconstructed without any loss of information. In contrast are lossy algorithms that, by forgoing the possibility of perfect reconstruction, can achieve even higher compression ratios. All of these methods try to exploit some property of the data under consideration. For example, Huffman-coding exploits non-uniform symbol probabilities whilst the MP3-format utilizes limitation of the human hearing system [4]. For our application we have a couple of factors that limit available methods:

1. The compression algorithm should be well adapted for implementation on FPGAs. A suitable algorithm should have constant memory requirement, be sufficiently fast, and at the same time limiting FPGA resource utilization. This means that the realized design should only use a small part of the logic and memory elements available on the FPGA.

2. The algorithm must operate on a data stream, as opposed to the complete

signal. Thus the data should be processed as the data is received, using limited buffering.

3. No configuration parameters which depend on the characteristics of the signal should be required. The large number of detector channels in the experiments implies that any calibration or configuration of the compression quickly becomes prohibitive.

4. Minimal reconstruction errors of signal pulses. The input data consist of aperiodic signal pulses and electronic noise. The pulses need to be recorded as accurately as possible, whilst the remaining data only need to be recorded to facilitate so-called base-line corrections.

The first requirement hampers the use of any algorithm that employs floating point arithmetic, multiplication or division, as these operations require large circuits to realize. The third requirement removes the possibility to use Huffman encoding with precomputed probability tables or methods based on vector quantizations. Vector quantization methods store the difference between the signal and one or more reference signals [2], a technique which has previously been applied for similar applications [5].

1.4 Previous work

Difference predicted trace compression (DPTC) is a lossless compression system that has already been designed for the compression of front-end detector signals [6]. The authors found that for their system the dominating storage cost was not the pulses themselves, but rather the signal noise. The aim is to build upon their work and achieve a greater compression ratio by introducing lossy compression of the parts of the signal where noise dominates. Previously, this has been achieved by zero suppression [5, 7], which essentially cuts any part of the incoming data that is not part of a pulse. The drawback is that this requires detection and localization of the pulse to decide what to cut. A less intrusive method would be to downsample the regions outside the pulse. Similar ideas have been presented for the purpose of time-series comparison [8, 9]. These methods create signal approximations by adaptively replacing series of samples with a single sample. Ideally, this method replaces long sequences of samples in regions of little-to-no change while keeping more samples in regions of rapid change. These downsampling methods are by themselves not compression schemes, as they lack the ability to reconstruct the original signal, but adaptive downsampling could be used to derive a compression system by also recording the downsampling ratios to facilitate reconstruction.

A method that bears some resemblance to the above idea is to compress the data after applying the Discrete Wavelet Transform (DWT); see references [10, 11] for an introduction to DWT. This method has been applied for trace compression, and of particular interest is its use in the ALICE experiment at CERN [12]. In the discrete wavelet transform domain, the signal is described by a series of approximation

coefficients and several levels of detail coefficients. Each level-of-detail coefficient describes the presence of features of a specific size. The first detail coefficients describe feature sizes of roughly two samples, the next level four samples et cetera, thus level $n+1$ only contains half the number of coefficients as level n. Compression is then achieved either by lossless compression of these coefficients, or by selective removal of them. The latter is in principle very similar to adaptive downsampling, which also selectively removes details of specific sizes, thus making the DWT very interesting. An implementation based upon DWT would also have the benefit of being analyzable using wavelet theory [10, 11]. Despite these similarities and advantages it was decided against using DWT for the following reason. Computation of the DWT requires a large number of decimal number multiplications, which are very expensive to realize on FPGAs. The size of the DWT implementation and the necessary compression logic would probably become too large for this application. Efficient FPGA implementations have been described [13], but all of these rely on extensive pipelining[1], and complicated pipeline folding[2] to achieve a reasonable trade-off between speed and area usage. The authors have not made their implementations freely available, implementing DWT would thus entail duplicating their efforts. Another argument against DWT is that it might be incompatible with DPTC as it is designed to compress sample series and not detail coefficients.

1.5 Thesis outline

This thesis describes the development of a compression algorithm based upon adaptive downsampling that satisfies the requirements outlined in previous sections. Further we answer the following questions:

1. How does one select the downsampling ratio under the constraint that signal pulses should be stored at full amplitude and time resolution with limited information loss?

2. How should the downsampling information be recorded? Such information is required to reconstruct the original signal. If this information is encoded too spaciously, no data-size reduction will be achieved.

3. How does such a system compare to DPTC?

4. Can the eventual system be combined with DPTC?

Chapter 2 presents the proposed compression routine, a software implementation of said algorithm and the performance characteristics of the system. In Chapter

[1] Pipelining - The practice of splitting complex computations into smaller chunks which can be executed in parallel. The principle is the same as assembly line construction of cars where each station/stage handles one small task.

[2] Pipeline folding - Pipeline folding involves reusing the same pipeline for different calculations by slightly altering how the data travels through it. This would be equivalent of using the same car assembly line to make two different kinds of cars.

3, the FPGA and software implementations are presented and the development process described. Results, as for example compression performance, are presented in chapter 4. Finally chapter 5 contains a brief discussion and possible future work.

2

Algorithm description

This chapter provides a detailed description of the compression algorithm. First a brief description of possible compression methods, together with the proposed scheme. This is followed by two sections detailing the main components of the system, the compressor and the standard deviation estimator. One of the first decisions one has to make when designing a compression system is whether to design a lossy or lossless algorithm. The former achieves compression by selectively discarding parts of its input data, but it comes with the challenge of deciding what data to remove. In the latter class one finds DPTC, a preexisting system for trace compression which reduces data size by employing an alternate on-disk data representation. Further improvements of the method were deemed to have diminishing returns, especially as the authors of DPTC found that the dominating cost was representation of the noise. Thus finding a way to deal with the noise became the primary focus.

To reduce the cost of noise it was decided to conduct data reduction, that is, partial signal removal in regions where noise dominates. More specifically, by conducting adaptive downsampling. Downsampling is the process of removing signal samples and/or replacing groups of samples with a fewer number of samples. The algorithm downsamples by taking groups of samples and replacing them with their sum. Given that the length of the groups is known, one can calculate the average sample value of the group. These group lengths are variable, hence the name adaptive, and chosen such that they are small in regions with significant signal content and longer in regions dominated by noise. The effect is that the system will not attempt to represent the exact details of the noise, but instead only report its average. To choose the length of group j, denoted R_j, the estimated standard deviation of the noise is utilized to decide whether signal deviations are significant. A simplistic example and an overview of the compression system is provided in figures 2.1 and 2.2. A description of the depicted black boxes is provided in the following sections. First the rules governing R_j are discussed, as these constitute the foundation of the design. Then follows a description on how to choose R_j while adhering to these rules. Once the logic behind R_j is known, the problem of noise standard deviation estimation and how this can be used for significance testing is addressed. Finally, the issue of data encoding is described and it is shown how DPTC can provide an efficient data representation.

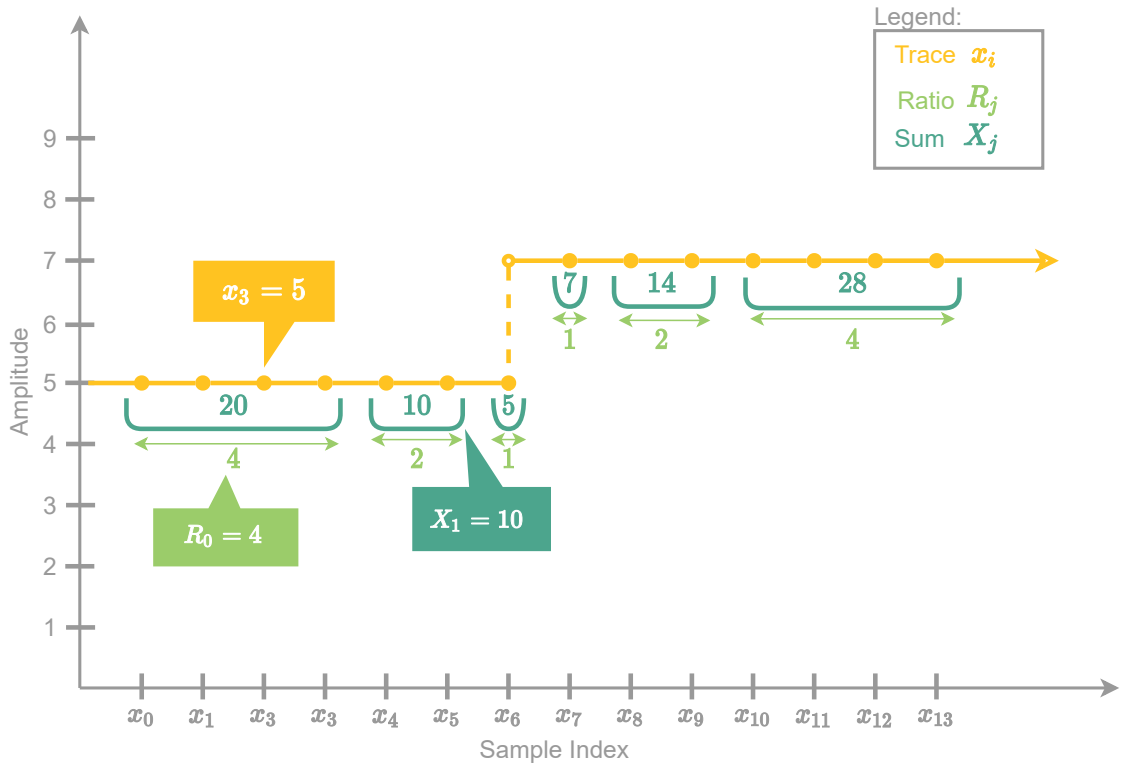

Figure 2.1: Illustration of adaptive downsampling of a short step function. The dots (yellow) is the input trace and the arrows (light green) and cups (dark green) indicate the system output, with downsampling ratio R_j and group sum X_j, i.e the sum of the R_j samples that is grouped together in group j.

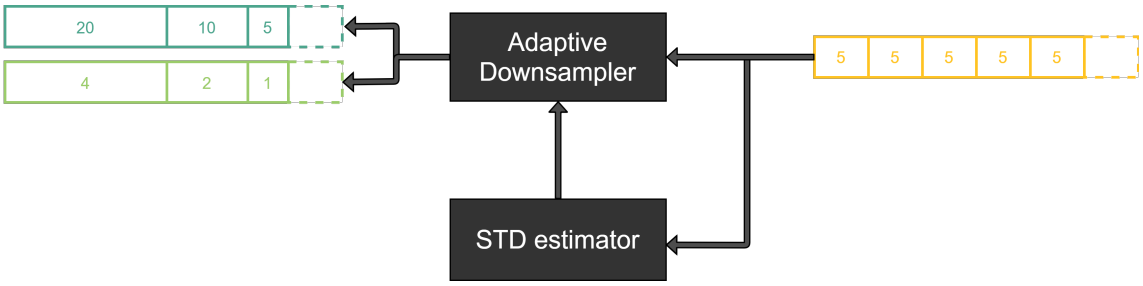

Figure 2.2: Schematic overview of the proposed system. The series x_i to the right denotes the original sample stream, R_j is the downsampling ratio and X_j is the sum of R_j consecutive samples. Note that this overview lacks the final encoding step for X_j and R_j.

2.1 Principle of operation

The first point to address is the rules governing R_j. The length, in samples, of each group is a power of two, ranging inclusively from 1 to $R_{\max} = 2^{L_{\max}}$. This size will from now on be referred to as the downsampling ratio and R_{max} as the maximal downsampling ratio. The maximal downsampling ratio is a user-defined parameter that affects performance, memory and hardware requirements, which will be addressed later. With R_j denoting the downsampling ratio of group number j, then only three possibilities are allowed for group $j + 1$:

$$R_{j+1} = \begin{cases} 2R_j \\ R_j \\ \frac{1}{2}R_j \end{cases} \tag{2.1}$$

$$1 \leq R_{j+1} \leq R_{\max} \tag{2.2}$$

These restrictions are enacted to aid performance and reduce implementation complexity. Limiting the change of downsampling ratio between groups to three possibilities reduces the coding overhead as only these steps need to be distinguished and encoded. If arbitrary ratios were allowed, then the selected ratio would have to be saved for each group. The limited step size smoothes the changes of resolution by hindering abrupt ratio changes before and after a sample that is deemed significant. Limiting R_j to a power of two simplifies hardware implementation whilst ensuring sufficient system response time to changes by reducing the number of steps in the range from $R_{\max}$ to 1. The rate at which R_j is allowed to change governs the memory or buffering requirement of system. Slower rate of change implies that more time is required to react to changes, thus a greater number of future samples must be inspected to decide the current downsampling ratio.

To select R_j that adhere to the enacted restrictions, the system splits the problem into two steps. The first is a group-local decision where the system decides which downsampling ratios would be acceptable for that given group based on the sample values and the estimated noise standard deviation alone. The next decision is on the macroscopic level. Given a series of these local restrictions, the system picks downsampling ratios so that equation 2.1 is never violated. First consider the local decision, or rather decisions, since the system needs to investigate if all ratios, two, four, eight and so on, are acceptable. These decisions are produced by the functions $f_2, f_4 \ldots$ which depend on the trace samples $x_i, x_{i+1} \ldots$ included in that group and the standard deviation of the noise σ. Of course, if it is decided that a ratio of 2 is not acceptable then ratios $4, 8 \ldots$ must also be disallowed. To express this formally some logical notation is needed, $\neg$ is used to denote logical negation and $\wedge$ denotes logical conjunction, colloquially known as the *and* operator. The local decision is

then computed as follows:

$$f_k : \mathbb{Z}_+^k, \mathbb{R} \to \{0, 1\} \tag{2.3}$$

$$C(i) = \begin{pmatrix} f_2(x_i, x_{i+1}, \sigma) \\ f_4(x_i, x_{i+1}, x_{i+2}, x_{i+3}, \sigma) \\ \vdots \\ f_{R_{\max}}(x_i, x_{i+1}, \ldots, x_{i-1+R_{\max}}, \sigma) \end{pmatrix} \qquad T(i) = \begin{pmatrix} C_1(i) \\ C_2(i) \wedge T_1(i) \\ C_3(i) \wedge T_2(i) \\ \vdots \end{pmatrix} \tag{2.4}$$

The meaning of the above definition is that samples $\{x_3, x_4, x_5, x_6\}$ can be grouped together if and only if $T_2(3) = 1$ which in turn requires $T_1(3) = 1$. The subscript is used to indicate a specific row of the vector $T(3)$. As a result, a downsampling ratio of 4 is acceptable only if ratio 2 is acceptable. What constitutes an acceptable downsampling ratio, i.e, the functional form of function 2.3 will be discussed in the next section. For now it is sufficient to note that it depends on the the estimated standard deviation of the signal noise denoted by σ. The function f_L should ideally be designed such that under the assumption that the signal is contaminated by zero mean additive Gaussian noise, f_L is true if the next 2^L samples contain nothing but noise.

The series $T(1),\ T(2), ...$ constitutes the basis of the macroscopic decision, i.e, the selection of $R_1,\ R_2, ...$ whilst adhering to restrictions 2.1 and 2.2. The system shall also be correct in the sense that the chosen step sizes never result in an invalid downsampling ratio being used. In a more compact form: if ratio R_i is used for sample x_k then it holds $T_{\log_2(R_i)}(k) = 1$. The implication of correctness is that the system must be non-casual, that is, it must have access to future samples. Consider the case when the system considers $R_j = 16$, to make a decision it needs to confirm that $R_{j+1} = 8$ is valid, which implies checking if $R_{j+2} = 4$ is okay. The worst case is when $R_j = R_{\max}$ is being considered, requiring a sample-look-ahead of:

$$\sum_{l=0}^{\log_2(R_{\max})} 2^l = 2^{\log_2(R_{\max})+1} - 1 = 2R_{\max} - 1. \tag{2.5}$$

The implication of equation 2.5 is that to select $R_{\max}$, the next $2R_{\max} - 1$ samples must be considered in order to guarantee correctness. Further, evaluation of $T(2R_{\max} - 1)$ requires another $R_{\max} - 1$ samples, due to the definition of f_l, thus the total required look-ahead is $3R_{\max} - 2$.

Before examining the details of the proposed system it could be educational to consider an example. Figure 2.3 shows a trace together with its reconstruction after compression. As expected, regions dominated by noise are downsampled and thus the noise is largely removed, whilst the pulse is reasonably preserved. Figure 2.4 shows how the downsampling ratio is varied during the pulse.

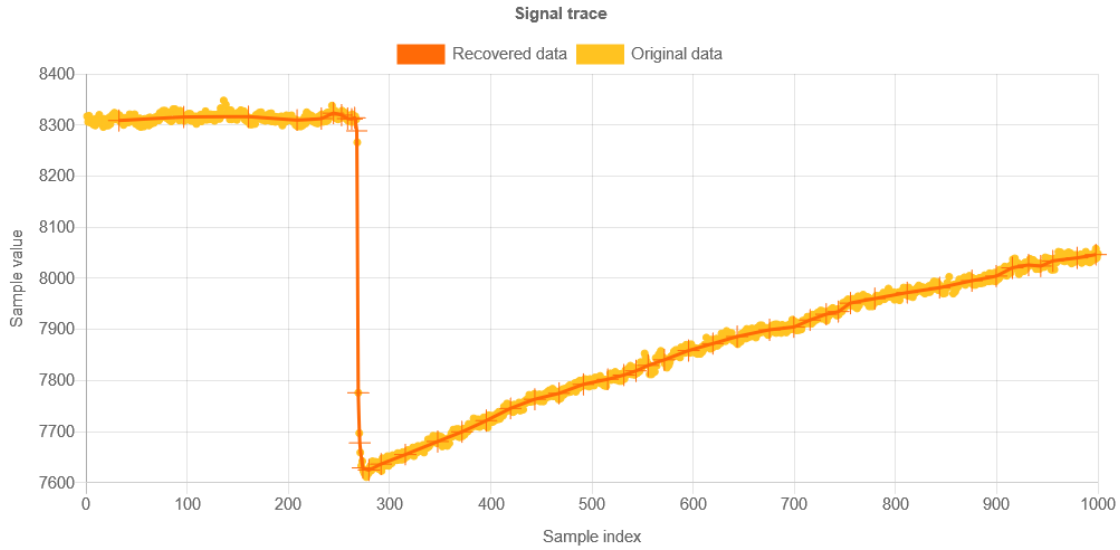

Figure 2.3: Example test trace, before and after being downsampled. The plot is produced by our public web utility see chapter 3.2.1

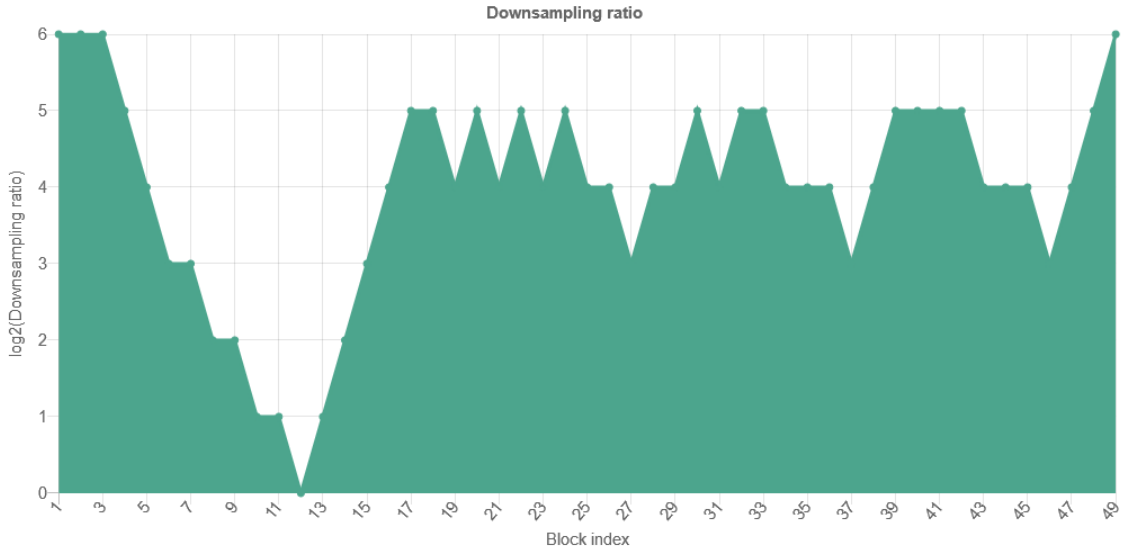

Figure 2.4: Illustration of downsampling ratio selected for the trace in figure 2.3. The plot is produced by our public web utility, see chapter 3.2.1.

2.2 Compressor design

In this section the details of the Adaptive Downsampler block of figure 2.2 are described. Figure 2.5 presents a schematic overview of this component. The main components are two First-In First-Out (FIFOs) data structures: the trigger FIFO which contains the sequence $\{T(i), T(i+1), \ldots, T(i+2R_{\max})\}$ and the two sample FIFOs which, together, hold the samples $\{x_i, x_{i+1}, \ldots, x_{i+3R_{\max}}\}$. The purpose of the FIFOs are two-fold. Firstly, they guarantee a sufficient look-ahead for the computation of R_i under the correctness constraint. Secondly the FIFO-based design reduces the length of the critical paths when realized on an FPGA by allowing pipelining and thus increase the achievable clock frequency of the design.

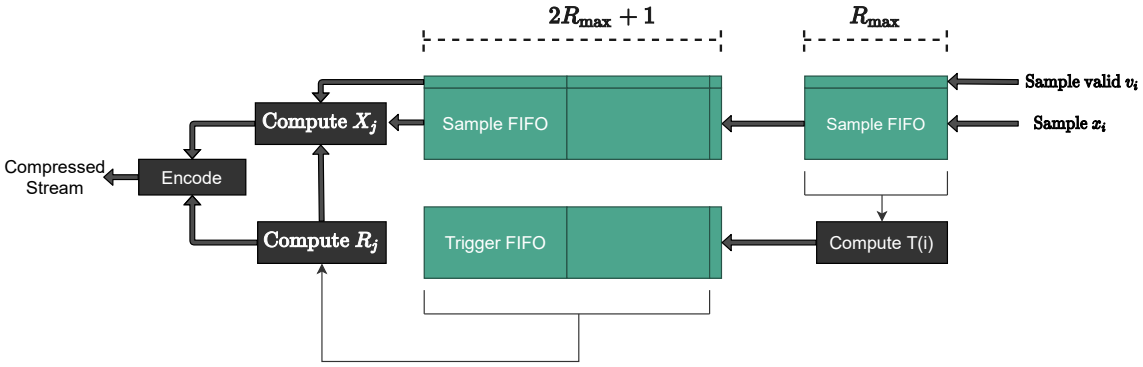

Figure 2.5: Schematic of the Adaptive Downsampler block in shown in figure 2.2. The dashed lines indicate the depth of the FIFOs. The signal v_i indicates if x_i is a valid sample and is used for FIFO flushing.

The selection of R_{j+1} is quite simplistic as constraint 2.1 implies that only three possibilities exist for a given R_j. The fundamental rule is that the system will select the highest allowable ratio under the constraint that the decision will not lead to the correctness property being violated now or in the future. To aid the decision a function is designed, called $Down(R_j)$ such that $Down(R_j) = 1$ implies that selecting the ratio R_j guarantees that the correctness property will be violated at some point and thus forcing the choice of $\frac{R_j}{2}$ instead. Guaranteed correctness violation occurs if the the downsampling ratio is reduced at every step but still results in a violation. Clearly this is as fast as the system can respond under constraint 2.1 and the violation is indeed unavoidable after picking R_j. To evaluate the function $Down$ for argument 32, one first needs to check that ratio $32 = 2^5$ is acceptable for the next 32 samples, thus checking $T_5(1), T_5(2), T_5(3), \ldots, T_5(32)$. Next one checks whether ratio 16 is acceptable for the following block, given that ratio 32 is chosen for the current block, that is, samples 33 to 48 are checked. The process then repeats for each following block whilst halving the ratio at every step until ratio 2 has been checked. Figure 2.6 illustrates the process of evaluating this function for different input arguments.

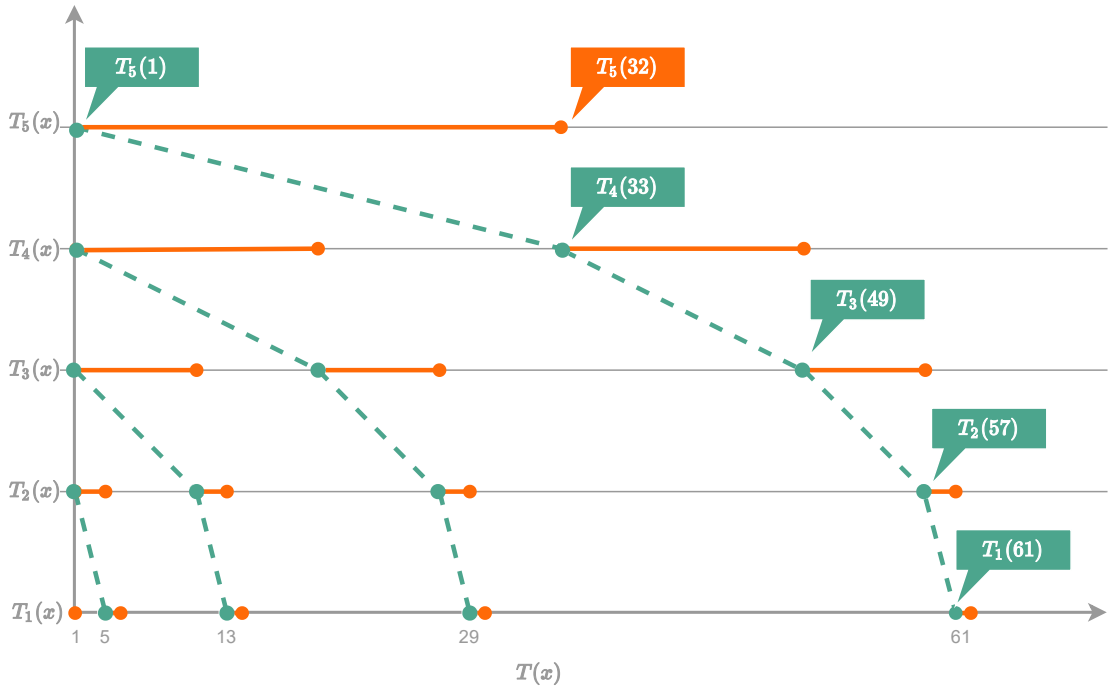

Figure 2.6: Illustration of $Down(R_j)$. Each dashed (green) line represents an evaluation of $Down(R_j)$ for a specific R_j. The top curve represents $Down(32)$. The solid (orange) lines indicate which $T_k(x)$ are required to compute the function.

Definition 2.6 (below) provides a formal formulation of the *Down* function.

$$\text{Down}(R_j) = \neg \left(\bigwedge_{k=2}^{(1+\log_2 R_j)} \bigwedge_{n=0}^{(R_j-1)} T_{k-1} \left(n + 1 + \sum_{l=k}^{\log_2 R_j} 2^l \right) \right) \tag{2.6}$$

Using the fact that the summation in the above equation is a geometric sum, it simplifies into:

$$\begin{aligned} \sum_{l=k}^{\log_2 R_j} 2^l &= \sum_{l=0}^{\log_2 R_j} 2^l - \sum_{l=0}^{k-1} 2^l \\ &= 2^{1+\log_2 R_j} - 1 - (2^k - 1) \\ &= 2R_j - 2^k. \end{aligned} \tag{2.7}$$

Using the closed form formula 2.7 above, one can remove the summation in definition 2.6 and arrive at:

$$\text{Down}(R_j) = \neg \left(\bigwedge_{k=2}^{(1+\log_2 R_j)} \bigwedge_{n=0}^{(R_j-1)} T_{k-1} \left(n + 1 + 2R_j - 2^k \right) \right). \tag{2.8}$$

Using the function $\text{Down}(R_j)$ as defined in equation 2.8 the change of ratio is selected by trying each of the expressions below in order and then pick the first true statement.

1. $R_j > 1 \wedge \text{Down}(R_j) \implies R_{j+1} = \frac{R_j}{2}$
2. $R_j \neq R_{\max} \wedge \neg\text{Down}(2R_j) \implies R_{j+1} = 2R_j$
3. else $\implies R_{j+1} = R_j$

In plain English this logic can be expressed in the following way:

1. If the downsampling ratio is larger than one and the downsampling ratio must be decreased in order to not violate the correctness property, then go down.
2. Increase the downsampling ratio if possible given that it does not violate correctness and that the new ratio does not exceed the maximal allowable ratio.
3. If none of the above, then maintain the current downsampling ratio.

Id est, the system will attempt to maximize the downsampling ratio under the constraint of the correctness property.

2.2.1 Ratio selection

There are many possibilities for the design of the set of trigger functions f_k from definition 2.3. Two such possibilities will be presented below.

A conceptually simple possibility is that the next R_j samples can only be grouped together if the maximum absolute deviation is within K standard-deviations.

$$f_{R_j}(x_i, x_{i+1}, \ldots, x_{i-1+R_j}, \sigma) := \left[\max\left(|x_i - \bar{x}_i|, |x_{i+1} - \bar{x}_i|, \ldots, |x_{i-1+R_j} - \bar{x}_i|\right) \leq K \cdot \sigma\right] \tag{2.9}$$

$$\bar{x}_i = \frac{1}{R_j} \sum_{k=0}^{R_j-1} x_{i+k} = \frac{X_j}{R_j}$$

This definition has the interesting property that it limits the reconstruction error of any single sample to $K \cdot \sigma$. Unfortunately it requires a large number of differences to be computed. Under the assumption that $x_i, x_{i+1}, \ldots$ are all independent identically-distributed Gaussian random variables, the difference $x_i - \bar{x}_i$ will be dominated by the variance of the single sample x_i.

The limitations of definition 2.9 lead to the development of an alternate definition:

$$f_{R_j}(x_i, x_{i+1}, \ldots, x_{i-1+R_j}, \sigma) = \left[|\bar{x}_i - \bar{g}_i| \leq \frac{K \cdot \sigma}{\sqrt{R_j}} \right] \tag{2.10}$$

$$\bar{x}_i = \frac{1}{R_j} \sum_{k=0}^{R_j-1} x_{i+k}$$

$$\bar{g}_i = \frac{2}{R_j} \sum_{k=0}^{R_j/2-1} x_{i+k}.$$

The advantage of 2.10 is that only a single difference comparison is required. In addition the difference is between averages, thus the impact of the noise is reduced as R_j increases. The compensation factor $\frac{1}{\sqrt{R_j}}$ reflects that the expected variance of the difference decreases as R_j increases. Once again, assume the input to be independent identically-distributed Gaussian variables, $x_i \sim \mathcal{N}(0, \sigma^2)$,

$$\begin{aligned}
\mathrm{Var}(\bar{x}_i - \bar{g}_i) &= \mathrm{Var}\left(\frac{1}{R_j} \sum_{k=0}^{R_j-1} x_{i+k} - \frac{2}{R_j} \sum_{k=0}^{R_j/2-1} x_{i+k} \right) \\
&= \mathrm{Var}\left(\frac{1}{R_j} \sum_{k=R_j/2}^{R_j-1} x_{i+k} - \frac{1}{R_j} \sum_{k=0}^{R_j/2-1} x_{i+k} \right) \\
&= \frac{1}{R_j^2} \mathrm{Var}\left(\sum_{k=R_j/2}^{R_j-1} x_{i+k} - \sum_{k=0}^{R_j/2-1} x_{i+k} \right) \\
&= \frac{1}{R_j^2} \left(\sum_{k=R_j/2}^{R_j-1} \mathrm{Var}(x_{i+k}) + \sum_{k=0}^{R_j/2-1} \mathrm{Var}(x_{i+k}) \right) \\
&= \frac{1}{R_j^2} \sum_{k=0}^{R_j-1} \sigma^2 \\
&= \frac{1}{R_j} \sigma^2.
\end{aligned} \tag{2.11}$$

Hence, it is concluded that the standard deviation of the differences $\bar{x}_i - \bar{g}_i$ scales as $\frac{1}{\sqrt{R_j}}\sigma$.

None of the above definitions guarantees a downsampling ratio of one for pulses since no pulse detection is included. The reasoning is that general pulse detection is very hard under the constraint that the system should not require manual tuning. Instead the compression system can be integrated with an external trigger by updating definition 2.4. Assume that $T_{\mathrm{Ext}}(i) = 1$ indicates that the external trigger has detected an event at sample i, then the updated definition 2.12 guarantees that that sample is not compressed:

$$T_1(i) = C_1(i) \wedge \neg T_{\mathrm{Ext}}(i). \tag{2.12}$$

This allows the end user of the system to amend the automatic ratio selection with an external trigger, if needed.

An interesting case to consider is when $K = 0$, as the system then becomes lossless. In this limit, the algorithm will approximate the standard run-length compression scheme, but with a constrained range of acceptable run-lengths. The performance characteristics of this case has not been thoroughly evaluated as the lossless mode of operation isn't the primary focus of this work.

2.2.2 Bit encoding and compression ratio

Each sample of the raw signal is encoded using a fixed number of bits denoted by A_{bits}. To store an uncompressed signal consisting of N samples would require $N \cdot A_{\text{bits}}$ bits. The compression system produces a stream of number pairs, the downsampling ratio R_j, and the sum of the samples X_j. This sample sum is the result of adding together R_j integers, each A_{bits} bits long, thus lossless encoding of X_j requires $A_{\text{bits}} + \log_2 R_j$ bits. R_j is encoded using Huffman coding with a static Huffman table 2.1, computed from the distribution of steps over several test traces. Testing showed the system to be significantly more likely to keep the same compression ratio, thus the cheapest encoding is reserved for the case $R_{j+1} = R_j$.

Table 2.1: Huffman encoding of downsampling ratio steps.

Step	Bit representation
$R_{j+1} = R_j$	0
$R_{j+1} = 2R_j$	11
$R_{j+1} = R_j/2$	10

Assume that a signal, which is N samples long, is compressed to M blocks and let $C_{\text{Bits}}(R_j)$ denote the cost to encode the downsampling ratio of block j, then the achieved compression ratio is given by

$$C_{\text{Ratio}} = \frac{N}{M + \frac{1}{A_{\text{bits}}} \sum_{i=1}^{M} \left(\log_2(R_j) + C_{\text{Bits}}(R_j)\right)}. \tag{2.13}$$

2.3 Standard deviation estimation of the noise

The estimation of the standard deviation of the noise is a critical component of the system as compression is mainly achieved by eliminating noise from the stored signal. A too low estimate would result in poor compression whilst an overestimation would, in the worst case, remove significant parts of the pulses. Noise standard deviation (STD) estimation is achieved by exploiting the fact that pulses occur rarely and that the majority of the signal is noise. Thus, if a sufficiently small random subset of the input signal is taken, it is unlikely to contain a pulse. Noise statistics can then be computed from this set.

Computing the standard deviation from the definition is not viable since it requires the computation of a square root and raising to powers of two. Such operations are

expensive operations in FPGAs and ill-fitted for integer math. A measure of the variability that is similar but sacrifices differentiability for computational simplicity is the Mean Absolute Deviation (MAD):

$$\mathrm{MAD}(y) = \frac{1}{N}\sum_{k=1}^{N}|y_i - \bar{y}|. \tag{2.14}$$

MAD still is not perfect for FPGA implementation as it requires all the samples of the random signal subset to be available. Further, the mean of the signal is expected to change over time which will drive up the difference in definition 2.14. Instead an alternate statistic is proposed, which is schematically presented in figure 2.7. Here the sample average $\bar{y}$ is replaced with a moving average $\bar{x}_i$ of size N. The absolute deviation is computed continuously which in turn is randomly sampled to create a set of approximately independent differences. Given a low enough sampling probability, together with low-pass filtering, reduces the effect of sampling a pulse. The low-pass filtering is achieved by feeding the randomly selected absolute deviations through a moving average filter of size $N_{\mathrm{Deviation}}$ followed by an infinite impulse-response (IIR) low-pass filter defined as

$$L_i = \alpha L_{i-1} + (1-\alpha)d_i. \tag{2.15}$$

Ideally the low-pass filter would only consist of a moving average, but the memory requirement of such a filter scales linearly with $N_{\mathrm{Deviation}}$. Instead the IIR low-pass filter is used as an approximate solution, to extend the number of deviations measurements that affect the deviation estimate without increasing the memory usage.

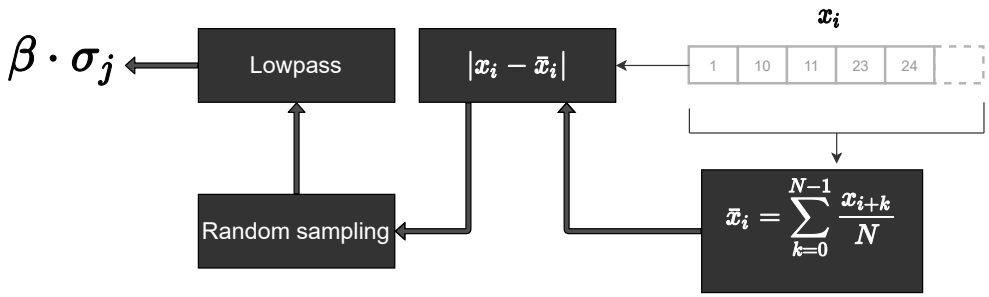

Figure 2.7: Standard deviation estimator block diagram. The absolute differences are continuously computed for each x_i of the input. Random sampling creates a new subset from these differences which is then passed through a low-pass filter.

2.3.1 Relation to conventional STD

The modified MAD definition can be related to the conventional standard deviation under the assumption that the samples x_i are identically distributed random variables.

The variance of the difference can be calculated as follows

$$\begin{aligned}\mathrm{Var}(x_i - \bar{x}_{i+1,N}) = \\ &= \mathrm{Var}(x_i) + \frac{1}{N}\mathrm{Var}(x_i) = \frac{N+1}{N}Var(x_i) \\ &\implies \sigma_{\mathrm{MAD}} = \sqrt{\frac{N+1}{N}}\sigma \end{aligned} \tag{2.16}$$

Thus as N increases the standard deviation of this difference approaches the STD of the noise. In the implementation N is chosen to 16 which results in $\sigma_{\mathrm{MAD}} \approx 1.03\sigma$

The expected value of the modified MAD statistic can be calculated as follows:

$$\begin{aligned} &p(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2} \\ &\int_{-\infty}^{\infty} p(x)dx = 1 \\ &\mathrm{E(MAD)} \approx \int_{-\infty}^{\infty} p(x)\cdot|x|dx = 2\int_{0}^{\infty} x\cdot p(x)dx = \\ &\sqrt{\frac{2}{\pi}}\sigma \approx 0.79788\cdot\sigma. \end{aligned} \tag{2.17}$$

These compensation factors can either be incorporated into the STD estimator or be accounted for when selecting the trigger level K of the system.

2.4 Decompression

Decompression is a short process involving two steps: Reconstructing R_j from the Huffman encoded steps and then reconstruction of the signal from X_j and R_j. The first task is achieved by assuming a value for R_1, which can either be transmitted fully or agreed upon beforehand, then the following values are computed from table 2.1 and the recursive application of equation 2.1.

Given the series X_j and R_j, the simplest reconstruction strategy is to set the first R_1 samples to $\frac{X_1}{R_1}$, then the following R_2 samples to $\frac{X_2}{R_2}$ et cetera. Whilst simple, the reconstruction will be overly jagged, and imply that more information is available than actually is the case.

A more sophisticated approach is to create the series:

$$\begin{pmatrix}\tilde{x}_j \\ \tilde{y}_j \\ w_j\end{pmatrix} = \begin{pmatrix}\sum_{k=1}^{j-1} R_k + \frac{R_j+1}{2} \\ X_j/R_j \\ \sqrt{R_j}\end{pmatrix}. \tag{2.18}$$

The series 2.18 represents each group is represented by a 3-tuple, $\tilde{x}_j$ represents the center of group j and $\tilde{y}_j$ the mean of the group. The last element, w_j is the groups

statistical weight, which reflect that the uncertainty of $\tilde{y}_j$ decrease with increased group size. Reconstruction is then done by fitting a suitable function to this series. This function needs to be carefully chosen by the experimentalist.

2.5 Boundary conditions

It has yet to be addressed how to handle the boundaries, that is, the last and the first groups. The simplest solution to this problem is to set the starting ratio to one and to modify the selection functions 2.4 so that any ratio that is larger than the number of remaining samples is disallowed, thus reaching the end exactly. The problem with this naïve approach is that for the majority of traces the compression would begin with the ratio ramping up, and then, as the end approaches, it would reduce the ratio back down to one or two in most cases. Thus, for short traces the storage costs would be dominated by these regions.

The initial boundary condition is resolved by freely picking the initial ratio and explicitly storing $\log_2(R_0)$ in the compressed stream. The initial ratio is selected by reusing the down function, equation 2.8:

$$\text{Initial}(R) = \neg\text{Down}(R) \wedge \neg\text{Down}(R/2) \wedge \neg\text{Down}(R/4) \wedge \ldots \tag{2.19}$$

R_0 is selected by finding the largest $R \in \{1, 2, 4, \ldots, R_{\text{MAX}}\}$ for which Initial(R) is true. The trailing boundary is solved by allowing the ratio of the last block to be larger than the remaining number of samples, i.e. even if just three samples remain of the trace, a ratio of four is still allowed. The ratio selection is modified so that if f_R in definition 2.4 can not be calculated due to a lack of samples, then it is considered to evaluate to true. Missing samples in the calculation of the sum X_i are replaced by zeros. This modification still allows perfect reconstruction of the last block given that L, the total number of original samples, is known. To recover the true length of the last block we use:

$$R_{\text{Last}} = L - \sum_{i=0}^{\text{Last}-1} R_i. \tag{2.20}$$

2.6 DPTC Integration

To achieve good compression it is essential to efficiently encode the pairs (X_j, R_j). An attempt was presented in section 2.2.2 where R_j was Huffman encoded, but X_j was left uncompressed. An important design goal of this project is to integrate with the preexisting compression system DPTC. Highly simplified, DPTC can be described as a lossless scheme that achieves compression by calculating the differences between two consecutive samples and then storing the differences with only the number of bits needed to represent them. For a more detailed explanation refer to the original article [6]. DPTC would be ideal to encode the series of X_j as, especially for large R_j, the differences between following groups are likely to be small.

The combination of the two is achieved by replacing our encoding step in figure 2.5 with DPTC, i.e. it is used to compress the stream of sample sums X_j. To achieve this, some slight modifications of DPTC are needed. Firstly DPTC must be made aware of the variable length header used to encode the downsampling ratio steps and include these headers in the output stream. Secondly, DPTC must also output the initial downsampling ratio. Figure 2.8 illustrates the bit encoding of the combined system for the previously-discussed example trace in figure 2.3; note that besides the addition of the ratio bits this is exactly the scheme used by the original DPTC.

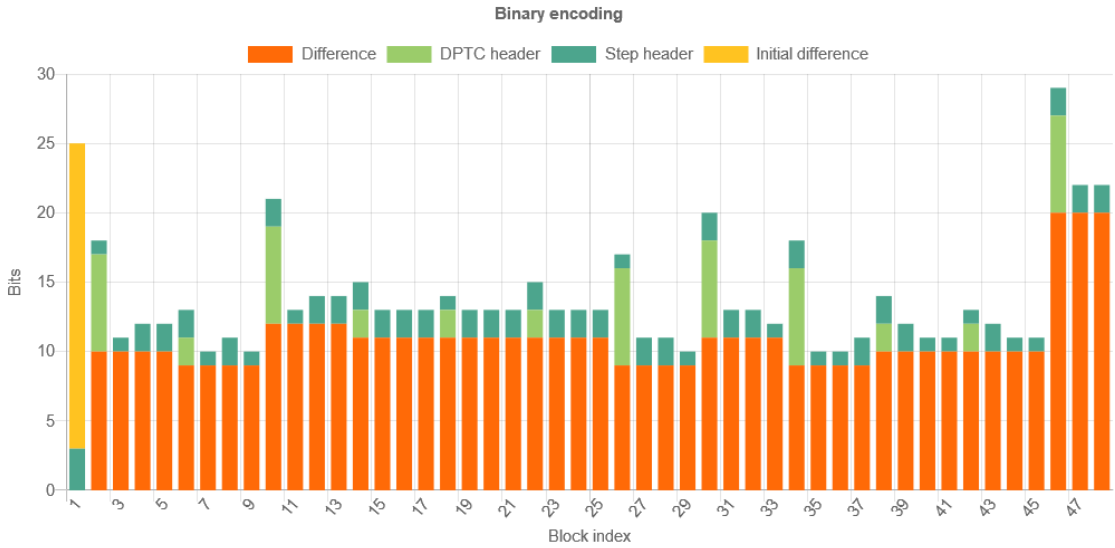

Figure 2.8: Binary encoding storage size of the trace in figure 2.3. The plot is produced by our public web utility, see chapter 3.2.1

.

To achieve good system performance, the DPTC differencing stage is also modified to account for downsampling ratio changes. Consider the example trace $[10, 10, 10]$, using only DPTC, this trace would result in the differences $[0, 0]$ which would be saved with a minimal amount of bits. Now assume that this trace, using our system, would be compressed into $[(10, 1), (20, 2)]$. DPTC will then calculate the difference $20 - 10 = 10$ which requires more bits to encode. To avoid this, a new difference method is introduced. Assume that p is the previous sample, s the current sample and d is the output difference, then the differences calculation depends on how the ratio was changed as follows:

1. If ratio has increased since the last block then $d \leftarrow s - 2p$
2. If ratio has decreased since the last block then $d \leftarrow 2s - p$
3. Otherwise $d \leftarrow s - p$

After each step, the previous sample is updated to $p \leftarrow s$. Using the above rules, reconsider the downsampled trace $(10, 1), (20, 2)$. First observe that the ratio used

increased between the two blocks, hence the first rule applies. Thus $d \leftarrow s - 2p = 20 - 2 \cdot 10 = 0$, $p \leftarrow 20$. This scheme ensures that the difference calculated by DPTC is as small as possible when the downsampling ratio changes. To reconstruct the compressed data, first recover the ratio change from the header before the difference and then solve for s in the above rules to recover the sample.

3

Software development

The aim of the project is to deliver a ready-to-use reference implementation of the compression algorithm described in chapter 2. The main component of this reference implementation is the FPGA firmware written in the hardware description language VHDL, but surrounding support software and utilities are also included. This chapter describes the produced software and the development process. In short, development consisted of three main phases:

1. Initial Matlab implementation,

2. C++ implementation,

3. VHDL implementation.

Producing three different software implementations can appear as an unnecessary effort, but this is offset by several factors. The implementations are listed in increasing implementation complexity. Writing VHDL code is a slow process. Compared to higher level languages, such as Matlab, VHDL, by necessity, introduces additional complexities to consider such as as logic clocking and timing to name a few. Initial development was instead done in Matlab as the reduced code complexity and excellent plotting functionality enabled rapid prototype development. Once one implementation was done, it could be used as a schematic for the next, thus allowing one to focus on implementation complexities and not on algorithm development. Each implementation was verified to agree with the previous by ensuring that for the same input data they produce the same output data. To find bugs and to improve confidence, for each implementation the author was switched. Hence each part of the algorithm has been understood and checked by two individuals.

The source codes for PC and FPGA (C++ and VHDL) are available for download at [14].

3.1 Matlab implementation

The Matlab implementation was solely written as a prototyping tool and is not released as part of the reference implementation. The goal was instead to quickly develop ideas for a possible algorithm which led to the proposed system. The soundness of these ideas was evaluated using both simple synthetic traces and the same real world traces used to test DPTC [6]. This version is replaced with a C++ implementation for two main reasons. Firstly, the Matlab source is hard to integrate with DPTC, which is written in C. Secondly, performance concerns due to the Matlab code having been written to mimic, as close as reasonable, the operation of the FPGA firmware. Code performance suffers without Matlab-specific optimizations, such as manual loop unrolling and vectorization, but addressing this sacrifices similarity with the final implementation. Finally, and most importantly, is that the DPTC reference implementation is written in C, so to test DPTC integration a C-compatible language is preferable.

3.2 C++ implementation

The C++ implementation is not just a stepping stone, but also an important testing tool and released as part of the reference implementation package. This component includes compression and decompression routines which utilize DPTC for encoding of the downsampled trace. DPTC support was achieved by modifying its sources to include the changes outlined in section 2.6. In addition, the source includes utilities for monitoring performance and calculating compression ratios. The end user can either use the source code directly or build the included command-line utility which provides a command-line interface to all of these functions. This tool was used to aid and verify the later VHDL implementation.

Software verification is limited to ensuring that it produces the same result as the VHDL and Matlab implementations for a large number of traces. More details on these test traces are given in the description of VHDL testing. The motivation is that extensive testing is only required for the VHDL implementation whilst the rest is verified by transitivity, i.e. by convincing that the implementations are equivalent.

3.2.1 Website

In addition to the command line tool, the C++ code was also utilized to create a demonstration website, available at `http://fy.chalmers.se/subatom/ads/compressor/`. This website allows a user to compress a trace of their choosing with any settings they require. The trace, reconstruction after compression and various performance statistics are graphically presented in the web browser. Figures 2.3, 2.4 and 2.8 are all produced by this website using the default settings and the example trace denoted "Real world Example". Figures 1.1 and 1.2 are produced by selecting "Real world Example 2" and setting the STD to 60.

The website was created by compiling the C++ code into web-assembly [15], which is

a standardized assembly language targeting modern web browsers. When a user connects to the website, his or her browser will download the web-assembly executable from the web server and then place the executable code into memory. The user interface and plotting functionality of the website are implemented in JavaScript, which in turn calls functions of the web-assembly binary for compression-related tasks. Thus the website and command line utilities not only execute the same compression algorithm but they share the same source code. Except for compiler or runtime defects it is highly unlikely that the website would produce a different result.

3.3 VHDL

The FPGA version of the algorithm consists solely of the compressor routine and is written in the hardware description language VHDL. All sources are written as to adhere to the VHDL-2002 standard, as the later revision, VHDL-2008, suffers from poor FPGA vendor support. All development was done using GHDL [16], an open source VHDL simulator, thus eliminating the need of vendor-specific toolchains during initial development.

Development was split into four main phases: component implementation and verification, system integration and verification, system testing and hardware testing. Component implementation consisted of splitting the algorithm into distinct components, such as sample FIFOs, trigger calculator, etc. and then implementing them in VHDL. Correctness of each component is verified by simulating it using GHDL whilst a test-bench script stimulates the module inputs and monitors its outputs. These test scripts serve as basic functionality checks and early bug detection. More complex functionality is realized by combination of more basic components.

The second phase, system integration, consists of integrating all components together to create the complete compression system, which is once again verified using GHDL. The emphasis of verification is to detect interface and logic errors rather than algorithmic errors. Algorithmic errors are captured in the third phase, system testing, in which the complete compressor is simulated using GHDL and fed numerous test traces and the compressed data is checked for errors. Setting the trigger level, K, to zero makes the algorithm lossless, thus simple comparisons between original and reconstructed traces are possible. Assume that the compressor produces block $X_i = 8, R_i = 4$, then the testing routine will decompress this block to $[2, 2, 2, 2]$ and then compare it to the original trace. If different, the test is considered to fail. The test trace collection consists of 22 artificial traces and 1723 real traces. The former consists of carefully crafted traces designed to provoke possible errors, for example traces which jump between the extreme representables given the number of bits allocated to each sample. The latter category consists of the real world examples bundled with the DPTC release [6].

The final step of VHDL implementation is synthesis and hardware testing. The current version of GHDL does not support design synthesis, i.e. transforming the VHDL code into a design fit for realization on an FPGA. Thus, for this step two

vendor toolchains were used, Xilinx ISE and Xilinx Vivado. The latter is Xilinx's newest toolchain targeting their latest generation of FPGAs, whilst ISE is their legacy offering. ISE was used to target the Spartan 6 FPGA and Vivado was used to target the ZYNQ-7000 System-on-a-Chip (SoC). These tools were used to characterize resource utilization and timing. An important goal was to find and optimize critical paths of the design in order to maximize the achievable clock frequency. Timing can be described as a signal race: each signal starts at a register; when the start signal sounds every single signal rushes along their signal paths through a maze of logic desperate to reach the finishing line, the next register, before a clock period has passed. If the logic is too convoluted, or the distance too great, signals will timeout before reaching their goal. The Xilinx tools was used to provide indication of signals failing the race, which in turn led to modification of the VHDL to either reduce logic complexity along the signal path or by pipelining, i.e split calculations over several clock cycles.

Limited on-hardware testing of the VHDL downsampling implementation was done on both FPGAs. The tests were done with the encoding step disabled, i.e. without DPTC, as this simplified the necessary data transfer protocols between the FPGAs and the host PC. The first test was using a Mojo v3 development board [17], equipped with a Spartan 6 FPGA. For this board, a simple implementation was created that allowed streaming downsampling of traces using the UART over USB interface of the board. Using this setup all artificial test traces were downsampled and verified to agree with the GHDL simulations. A second hardware test was done on a Digilent Arty Z7 board [18], equipped with a Zynq-7000 System-on-a-Chip . This SoC contains both an ARM processor and a FPGA on a single chip, with interconnects in-between to allow high-speed communication. Test traces were transferred from the host PC using Ethernet to the Zynq processor, which in turn transferred the trace to the FPGA for downsampling. Once completed, the transfer direction is reversed and the downsampled trace was transmitted back to the PC where the result could be verified. Compared to the Mojo test, this board allowed much higher data rates due availability of high speed interfaces. The purpose of testing on hardware is to verify that no unexpected behaviour occurs on hardware and that the GHDL simulations are indeed truthful to reality.

4

Results

In this chapter, the results of this work are presented, consisting of compression statistics, system characterization, VHDL design resource utilization and timing statistics.

4.1 Compression performance

The first and maybe most important statistic to consider is the compression performance, presented in table 4.1, which is a modification of table III from the DPTC-article [6]. All columns except the last two are replicated from the article. For each trace category, the following information is given: number of traces, number of samples per trace, estimated noise standard deviation and average bits per sample for DPTC only, for Adaptive Downsampling only and for the combined system. When using Adaptive Downsampling only, the stored trace is encoded as outlined in section 2.2.2, whilst in the combined system DPTC provides the output encoding. The two columns were produced by compressing each trace in the collection and then calculating the average number of bits per sample. For these runs the following settings were used: $K = 3$, $R_{\max} = 64$, $A_{\text{bits}} = 14$ and the standard deviation estimator was disabled and replaced with the category estimate in column six. The STD-estimator was disabled as the total trace lengths compared to the convergence time of the estimator was too short. The STD-estimator is designed to operate over much longer intervals and sample the ADC output, regardless of whether a trace is currently being compressed or not. Reconfigurating the STD-estimator to operate on shorter timescales and faster sampling was not an option, as for these short traces the assumption of events being rare is no longer valid. Thus a faster STD-estimator would characterize not only the noise, but also be affected by the signal.

Table 4.1 seems to indicate that Adaptive downsampling on its own greatly reduces the storage costs, but to draw this conclusion one also needs to consider that the system is lossy and any gains must be weighed against potential information loss. Requirements might dictate a more conservative trigger level of $K = 2$, as shown in table 4.2, which naturally leads to worse performance. Thus performance numbers presented herein only serve as an indication of what is achievable, but results will vary depending on a given application. What can be concluded is that in all cases

Label	Category	Details	Traces #	Samples #	σ	DPTC	ADS	ADS + DPTC
						— — — Bits/Sample — — —		
a		core signal	40	5000	2.16	3.89	1.70	0.79
b	γ in segmented BEGe	segment 1	40	5000	2.16	3.86	1.81	0.85
c		segment 5	40	5000	2.21	3.91	1.81	0.85
d		Ionisation chamber	200	200	71.2	9.16	1.84	1.53
e	n/γ discrimination	n-det, anode	200	200	4.88	5.36	2.67	1.72
f		n-det, cathode	200	200	6.20	5.71	2.48	1.59
g	position-sensitive	α-particles	50	1000	29.7	7.81	1.13	0.81
h	Si pin-diode	^{40}Ar	50	1000	6.36	5.58	2.28	1.35
i		no signal split	100	200	5.30	5.55	3.88	2.51
j	γ from ^{137}Cs	signal split 1:2	100	200	3.90	5.08	3.46	2.15
k	in $LaBr_3$	signal split 1:4	100	200	3.23	4.81	3.20	1.95
l		signal split 1:8	100	200	3.05	4.65	2.78	1.69
m		350V	100	600	0.25	1.67	1.90	0.62
n	cosmic μ in $LaBr_3$,	400V	100	600	0.25	1.67	2.22	0.74
o	varying HV of PMT	450V	100	200	4.28	5.55	4.55	3.01
p	cosmic μ in $LaCl_3$,	CAEN DT5730	100	400	3.88	5.00	2.50	1.49
q	different digitizers	CAEN DT5751	100	400	0.86	2.72	2.95	1.28
r		all values 0	1	1000	0	1.51	0.34	0.06
s	Flat traces	all values 10	1	1000	0	1.51	0.34	0.09
t		all values 100	1	1000	0	1.51	0.34	0.10

Table 4.1: Storage efficiency of DPTC and adaptive downsampling (ADS) for different trace categories with $K = 3$. The three right-most columns present the average number of bits per sample, i.e the total compressed size divided by number of samples in the trace. This table is a replication of table III presented in the DPTC-article [6], but adopted to demonstrate ADS performance.

both ADS and DPTC by themselves reduces storage costs, but their combination provides even greater gains. Assuming losses are acceptable, ADS+DPTC have the potential to provide a significant data size reduction compared to using only DPTC, this is illustrated by figure 4.1.

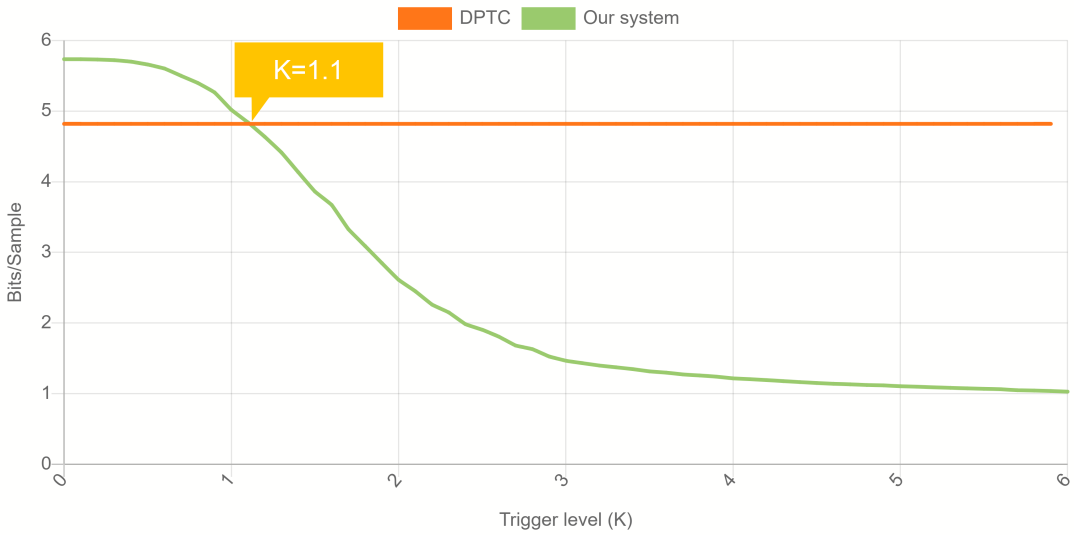

Figure 4.1: Sweep of trigger level for all test traces in table 4.1 except for the flat traces. The y-axes indicates the total average number of bits/sample for all of these traces after compression with ADS+DPTC. For $K > 1.1$ ADS+DPTC provides better compression than only using DPTC.

Label	Category	Details	Traces #	Samples #	σ	DPTC	ADS	ADS + DPTC
						— — — Bits/Sample — — —		
a		core signal	40	5000	2.16	3.89	4.80	2.08
b	γ in segmented BEGe	segment 1	40	5000	2.16	3.86	4.76	2.07
c		segment 5	40	5000	2.21	3.91	5.06	2.20
d		Ionisation chamber	200	200	71.2	9.16	5.41	4.08
e	n/γ discrimination	n-det, anode	200	200	4.88	5.36	5.48	3.06
f		n-det, cathode	200	200	6.20	5.71	5.42	3.06
g	position-sensitive	α-particles	50	1000	29.7	7.81	2.93	1.93
h	Si pin-diode	^{40}Ar	50	1000	6.36	5.58	4.77	2.60
i		no signal split	100	200	5.30	5.55	5.61	3.34
j	γ from ^{137}Cs	signal split 1:2	100	200	3.90	5.08	5.76	3.17
k	in $LaBr_3$	signal split 1:4	100	200	3.23	4.81	5.75	3.03
l		signal split 1:8	100	200	3.05	4.65	5.83	2.99
m		350V	100	600	0.25	1.67	3.74	0.95
n	cosmic μ in $LaBr_3$,	400V	100	600	0.25	1.67	4.41	1.13
o	varying HV of PMT	450V	100	200	4.28	5.55	5.80	3.58
p	cosmic μ in $LaCl_3$,	CAEN DT5730	100	400	3.88	5.00	5.86	3.00
q	different digitizers	CAEN DT5751	100	400	0.86	2.72	5.67	2.14
r		all values 0	1	1000	0	1.51	0.34	0.06
s	Flat traces	all values 10	1	1000	0	1.51	0.34	0.09
t		all values 100	1	1000	0	1.51	0.34	0.10

Table 4.2: Performance of DPTC and adaptive downsampling (ADS) for different trace categories with a reduced trigger level of $K = 2$ instead of 3. Reducing K increases system sensitivity making downsampling less likely. Thus the signal is recorded more accurately at the cost of on average allocating more bits per sample. The three right-most columns present the average number of bits per sample, i.e the total compressed size divided by number of samples in the trace.

4.2 Noise STD estimator characteristics

The noise STD estimator was tested by comparing its estimate to two conventional methods when executed on synthetic noise with a known STD. The first method is the cumulative STD, i.e. to compute the STD from the definition using all samples before the current sample. Whilst such a method would be accurate, it is not viable for real-time usage due to the large memory requirements. A more realistic method is windowed STD, which limits memory usage by computing the STD using the definition but only the last N samples are considered. Nonetheless, windowed STD is very expensive to compute as it requires the computation of square roots and that has higher memory requirements than the proposed STD estimator. Figure 4.2 illustrates how all three methods behave when used on generated noise with a known STD of ten. Using this figure some conclusions can be made. Firstly, the proposed method converges much slower than the other methods, but this is not necessarily a bad thing. The standard deviation estimator is designed to be connected directly to the continuously sampling ADC and do random sampling regardless of whether a trace is being recorded. In many use-case such an ADC will operate at frequencies around of 50–100 MHz and above. Thus even a filter that requires a hundred million samples to converge will have done so after a second or two of the system being turned on. See table 4.3 for example convergence time. Being this slow the STD estimator operates at a much larger timescales than the important signal pulses. If these signal pulses are rare enough then one can argue that it is the background noise that the STD estimator characterizes, and not the

signal pulses. For applications where this assumption is not valid one can still apply the compression system, but then the STD estimator needs to be replaced by an alternative way of providing the standard deviation of the noise.

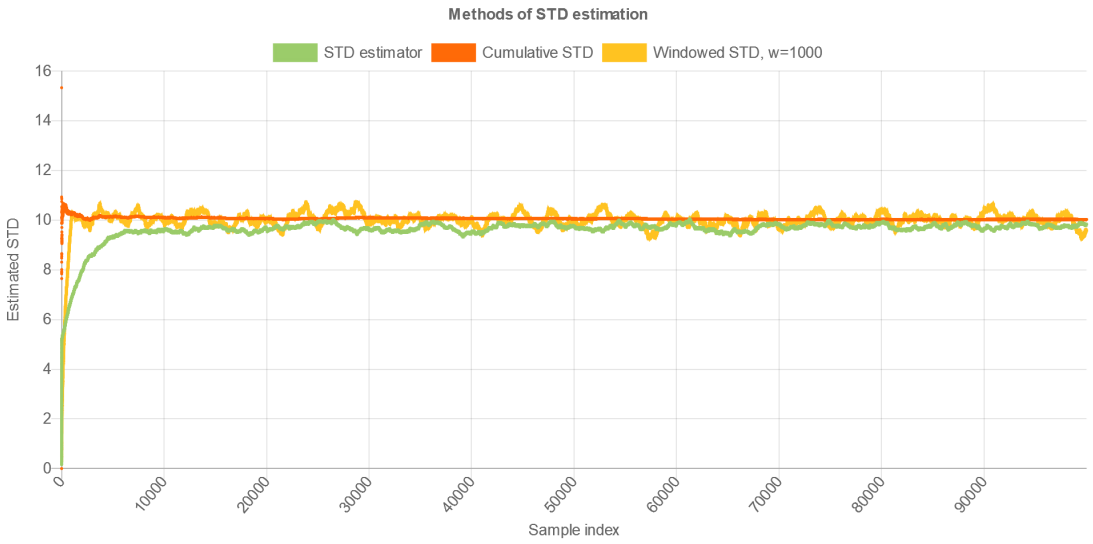

Figure 4.2: Comparison of STD estimation methods on synthetic noise with a known STD of ten. The green line indicates the STD estimator proposed in this thesis whilst the others represent different applications of computation using the standard deviation definition. The x-axis is the current sample index and the y-axis indicates the estimated STD after inspecting that number of samples, i.e. the y-value at $x = 10000$ indicates the estimated STD after the filter has been provided 10000 samples. Note that the STD estimators random sampling was disabled when generating this figure to make comparison easier. Enabling it would only result in the green line being further elongated along the x-axis with a factor equal to the inverse of the probability of to take a sample.

To try to answer the question of how rare the pulses need to be for the application of the STD estimator, a simple test was devised. Synthetic noise with a known STD was generated, and the a number of Gaussian pulses (see equation 4.1) was added to the noise to create a noisy pulse train.

$$f(x) = a \cdot e^{-\left(x^2/(2 \cdot c^2)\right)} \tag{4.1}$$

The STD estimator was then used on this data and the resulting STD was compared to the to the true STD of the noise, each generated trace's length was selected to

STD	Samples until convergence
1	14881
10	8799
100	10807

Table 4.3: STD estimator convergence times in number of samples when operated on synthetic noise.

Figure 4.3: Relative STD error as a function of pulse percentage. Data is averaged over 100 traces. The pulses was generated with function 4.1 with $a = 160$ and $c = 7$, the noise had a STD of ten.

account for the convergence time of the estimator. This experiment was repeated multiple time with a different number of pulses to understand the STD estimate error as a function of pulse content. Each pulse of the generated trace are identical, with fixed width (c) and amplitude (a), and equidistantly separated. Figure 4.3 shows the relative STD error of the estimator versus the signal pulse percentage. The pulse percentage is the number of samples of the trace which contains samples which are part of an Gaussian pulse. If a Gaussian pulse consisting of 100 samples is added to a trace of length 1000 then the signal pulse percentage is 10%. From these test it can be concluded that a very low pulse percentage is required for the STD estimator to be reliable, though the overestimation can be somewhat compensated by selecting selecting a smaller K. To improve the STD estimator one would need to add an outlier/pulse rejection filter to it, which actively reduces the risk of including pulses into the STD estimation process.

4.3 FPGA resource utilization and timing

The characteristics of the VHDL design was evaluated on four different FPGAs, which are presented in tables 4.4 and 4.5. The total resource utilization is quite low and several circuits could be realized on the same FPGA, in case of the Zynq-7000 the usage is less than 3% of available resources. LookUp Tables (LUTs) are configurable FPGA elements with a small number of inputs on which it can perform arbitrary logic functions. LUTs serve as the basic building blocks of FPGAs which are combined to realize complex logic. The Virtex-4 has significantly higher LUT usage, as its LUTs have only four inputs instead of six as the others, hence the functions realizable by a single Virtex-4 LUT are fewer. The two other major building blocks are memory elements, Flip-Flops (FF) and Block RAMs, of which only the former is shown in the table. Flip-flops are small memory elements which, depending on FPGA, are able to store just a few bits, often just one bit per Flip-flop. These small memory elements can be be interconnected to create larger memory arrays, with the possibility to intersperse these arrays with LUTs to implement arbitrary logic circuits. Block RAMs are much larger, fixed function memory arrays, able to store larger amounts of data, but often comes with the restriction that only one or two elements of the array can be accessed per clock cycle. Consider figure 2.5, the large sample FIFO can be realized using a block RAM as only the ends of the array is accessed, whilst the smaller Sample FIFO and the trigger FIFO has more complex access patterns and thus require another approach. Exactly how the different elements are implemented depends on FPGA and the decisions made by the FPGA vendor-supplied synthesizer tool.

The design can processes up to one ADC sample each clock cycle, thus the clock speeds presented in tables 4.4 and 4.5 is also the highest achievable sampling rate. For example, the Zynq-7000 achieves 244 MHz at $A_{\text{bits}} = 14$ and is thus able to handle data input flows up to $14 \cdot 213 \cdot 10^6$ bits/s or 373 MB/s. It is possible to further improve the maximal clock frequencies by optimizing the VHDL.

FPGA Family	Part number	Max frequency	LUT usage	FF usage
Zynq-7000	xc7z020-1	213 MHz	1329	1177
Spartan-7	xc7s75-1	215 MHz	1335	1177
Spartan-6	xc6slx9-2	100 MHz	1206	1123
Virtex-4	xc4vlx15-12	155 MHz	1741	1136

Table 4.4: Design performance and resource utilization on different FPGAs. Max frequency is the maximal achievable clock frequency of the design as currently implemented. LUT stands for lookup table. These are the elements that realize combinatorial logic in the FPGAs. FF are Flip-flops which are a type of FPGA memory unit. The design is synthesized with $A_{\text{bits}} = 12$, $R_{\text{MAX}} = 64$.

The current DPTC compressor/decompressor implementation restricts the choices of ADC widths and max ratio somewhat. These restrictions are very much a limitation of current implementation and not inherent to the algorithms them-self, thus if the

FPGA Family	Part number	Max frequency	LUT usage	FF usage
Zynq-7000	xc7z020-1	213 MHz	1437	1289
Spartan-7	xc7s75-1	215 MHz	1437	1289
Spartan-6	xc6slx9-2	100 MHz	1314	1227
Virtex-4	xc4vlx15-12	175 MHz	1882	1242

Table 4.5: Design performance and resource utilization on different FPGAs. The design is synthesized with $A_{\text{bits}} = 14$, $R_{\text{MAX}} = 64$.

need arises one can modified the source to remove these limitations. To strike a balance between performance and usability, the software DPTC implementation is limited to processing 32 bits data words at a time, the effect of this when combined with ADS is that the following must hold:

$$2 + \log_2(R_{\text{MAX}}) + A_{\text{bits}} < 32 \tag{4.2}$$

The above equation is a result of DPTC being limited to ADC sizes of 31 bits, with the addition of ADS, the output is extended by an additional $\log_2(R_{\text{MAX}})$ to account for the fact that the DPTC input is now a sum of samples. The two is the worst case size of the step header.

The DPTC VHDL implementation is even more conservative as it limited to emitting 32 bits per clock cycle, which, in the worst case, must fit all of the above and a DPTC long header.

$$A_{\text{bits}} + \log_2(R_{\text{MAX}}) + 2 + 2 + \text{ceil}(\log_2(A_{\text{bits}} + \log_2(R_{\text{MAX}}) - 4)) < 32. \tag{4.3}$$

Assuming $R_{\text{MAX}} = 64$, which is at the upper limit of a reasonable realworld value yields

$$A_{\text{bits}} + \text{ceil}(\log_2(A_{\text{bits}} + 2)) < 22 \tag{4.4}$$

Solving the above equation numerically yields a maximum ADC size of 16 bits. If larger ADC sizes are required one would need to slightly modify the output encoding step of the DPTC implementation to handle the potentially larger data overflows. As of current, these implementation specific requirements are not deemed to constitute significant restrictions in real-world scenarios, hence no effort has been made to remove these.

5

Discussion

Based on the results of this work, it can be concluded that the goals and requirements of the project have mostly been met. The motivation behind this statment will be provided in this section. First follows a discussion regarding the fulfilment of algorithm requirements as presented in chapter 1.3, after which interesting features and future work are addressed.

The first requirement, suitability of algorithm for FPGA use, can safely be deemed as fulfilled, as the system can be realized with integer operations such as bit shifts, differences and a few multiplications. The heavy reliance on FIFO data structures makes it ideal for FPGAs, as any issues could easily be solved by lengthening queues and separating calculations over several steps. The increased latency introduced by these extra steps is of minimal concern for this application. Clearly this approach also satisfies the second requirement of streaming compression, but the requirement of minimal buffering is less obvious. In chapter 2 it was shown that the required FIFO length, and thus latency, scales linearly with the the allowable max ratio. The end user can freely choose the max ratio to strike a balance between resource utilization, latency and achievable compression ratios. In addition to the max level, the user needs to select number the of ADC bits and the trigger level K. Even with these, the third requirement, to have no signal dependent configuration parameters, is deemed to be satisfied as these are not calibration settings. Users willing to conduct system calibration can bypass the standard deviation estimator and provide their own estimate, thus gaining a more predictable system.

The fourth statement, minimal reconstruction errors of signal pulses is the only requirement for which fulfillment is difficult to establish. An answer requires knowledge on what constitutes minimal reconstruction errors. Of course, setting the max-ratio to one achieves this, but fails to achieve data compression. If one guarantees no downsampling of the event pulse, then this can be accomplished with the caveat of requiring pulse detection. This is hard to do in a system agnostic of signal properties. If guaranteed perfect pulse reconstruction is required, then the compressor should be integrated with an external trigger as proposed in equation 2.12. The answer is further complicated by the fact that what constitutes signal versus noise depends on the chosen application. Ideally, each real-world trace should be analyzed before and after compression with the same methods employed by the producer of

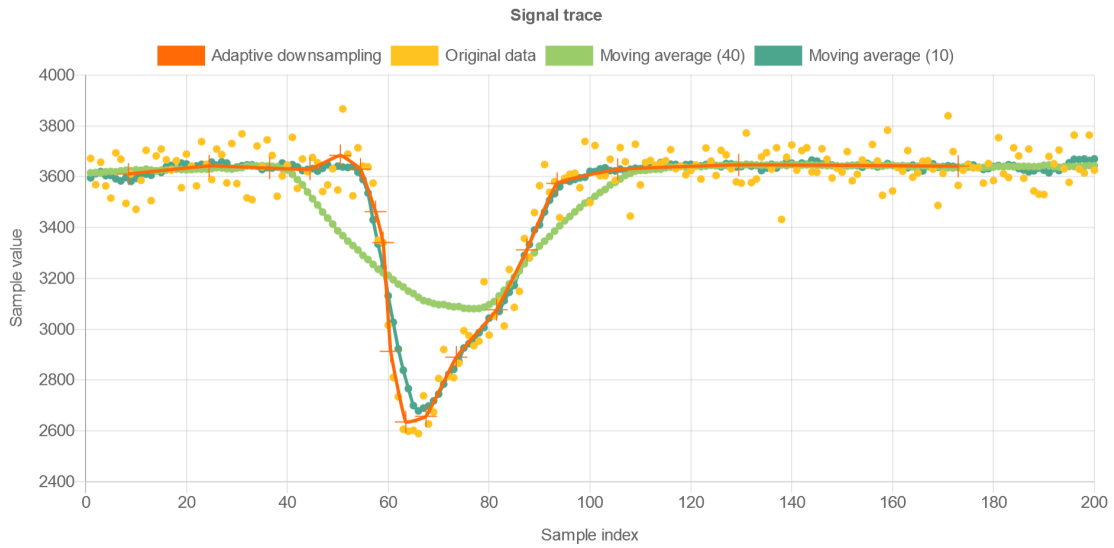

Figure 5.1: Illustration of a signal together with three different filtered versions, adaptive downsampling with the STD fixed to 200 and moving averages of two different lengths. Do note that a moving average does not provide data compression as the number of samples remains the same.

that trace to understand possible information loss. Unfortunately the effort to do so would be prohibitive and is considered outside the scope of this project. Ultimately what are acceptable errors must be decided by the user with help of the presented material.

It is interesting to also consider other possible applications for this method of adaptive downsampling, for example data analysis. Adaptive downsampling can be thought of as an averaging filter where the averaging window length is modulated to preserve level changes. Removing the STD estimator then allows the user to set the STD, which regulates the degree of signal smoothing. Figure 5.1 illustrates how adaptive sampling compares to two different sizes of moving averages. The moving average is calculated by replacing each sample with the average of the sample together with its N closest neighbours. From this image one could possibly argue that Adaptive downsampling smoothes the signal whilst preserving the general signal shape better than the moving average. Adaptive downsampling also reduce the number of trace samples thus potentially making function fits easier. A reduced number of samples can speed up function fitting. The impact of noise on the fit is somewhat reduced, as the downsampling process reduces stored signal noise. Regardless this paragraph only serves as an curious remark, as this application has not been thoroughly studied.

Future development work of the compression system should focus on testing the complete system in real-world scenarios. That is testing the system installed on a front-end card complete with the STD estimator and ADCs. With this setup one would then evaluate how the compressed and reconstructed traces compare

to to the raw trace with an emphasis on how compression affect possible function fittings. Unfortunately, for the traces employed in testing the system, the functional form used for fitting the pulse shape is unknown, and thus evaluating the effect of compression is hard. This problem is further aggravated by the fact that what constitutes acceptable loss is not known for each trace. Given that the compression system is lossy and that the possible information loss is governed by K from equation 2.10, such testing is anyhow required by the end user to find an acceptable K for their specific application.

Bibliography

[1] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[2] K. Sayood, *Introduction to Data Compression*. Elsevier Inc., 2012.

[3] R. C. Gonzalez and R. E. Woods, *Digital Image Processing (3rd Edition)*. Pearson, 2007.

[4] S. Shlien, "Guide to MPEG-l Audio Standard," *IEEE Transactions on Broadcasting*, vol. 40, no. 4, pp. 206–218, 1994.

[5] 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," in *IEEE Transactions on Nuclear Science*, vol. 58, no. 4 PART 1, 8 2011, pp. 1738–1744.

[6] G. Bruni and H. T. Johansson, "DPTC—An FPGA-Based Trace Compression," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 1, pp. 189–197, 2020.

[7] A. Werbrouck, F. Tosello, A. Rivetti, G. Mazza, P. De Remigis, D. Cavagnino, and G. Alberici, "Image compression for the silicon drift detectors in the ALICE experiment," *Nuclear Instruments and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 471, no. 1-2, pp. 281–284, 2001.

[8] C. C. Chia and Z. Syed, "Using adaptive downsampling to compare time series with warping," in *Proceedings - IEEE International Conference on Data Mining, ICDM*, 2010, pp. 1304–1311.

[9] M. Krawczak and G. Szkatuła, "An approach to dimensionality reduction in time series," *Information Sciences*, vol. 260, pp. 15–36, 3 2014.

[10] A. B. Romeo, C. Horellou, and J. Bergh, "N-body simulations with two-orders-of-magnitude higher performance using wavelets," *Monthly Notices*

of the Royal Astronomical Society, vol. 342, no. 2, pp. 337–344, 06 2003. [Online]. Available: https://doi.org/10.1046/j.1365-8711.2003.06549.x

[11] ——, "A wavelet add-on code for new-generation n-body simulations and data de-noising (jofiluren)," *Monthly Notices of the Royal Astronomical Society*, vol. 354, no. 4, pp. 1208–1222, 11 2004. [Online]. Available: https://doi.org/10.1111/j.1365-2966.2004.08303.x

[12] 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," *Nuclear Instruments and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 527, no. 3, pp. 580–590, 7 2004.

[13] C. F. Hsieh, T. H. Tsai, C. H. Lai, S. C. Yi, and M. H. Yen, "Implementation of an efficient DWT using a FPGA on a real-time platform," in *Second International Conference on Innovative Computing, Information and Control, ICICIC 2007*. IEEE Computer Society, 2007.

[14] "Adaptive Downsampling of traces." [Online]. Available: http://fy.chalmers.se/subatom/ads

[15] "WebAssembly Specification — WebAssembly 1.1," accessed: May 20, 2020. [Online]. Available: https://webassembly.github.io/spec/core/

[16] "GHDL Main/Home Page," accessed: Jan 27, 2020. [Online]. Available: http://ghdl.free.fr/

[17] "Mojo V3 | Alchitry," accessed: May 20, 2020. [Online]. Available: https://alchitry.com/products/mojo-v3

[18] "Arty Z7: APSoC Zynq-7000 Development Board for Makers and Hobbyists - Digilent," accessed: May 20, 2020. [Online]. Available: https://store.digilentinc.com/arty-z7-apsoc-zynq-7000-development-board-for-makers-and-hobbyists/