Data Compression Techniques For Tracking Chamber Digitisations

THE UNIVERSITY of MANCHESTER

Andrew N. Pickford

Department of Physics and Astronomy

A thesis submitted to the University of Manchester for the degree of Master of Science in the Faculty of Science and Engineering.

September 25, 1995

Contents

List of Figures

List of Tables

Abstract

Data compression techniques for drift chamber pulse digitisations were investigated. The lossless compression methods studied included Huffman and Arithmetic encoding of both unlinearised digitisations and the difference of samples of the digitisations. The best lossless compression method for a single digitisation was Huffman encoding which achieved an average reduction of 35.3 percent in the size of the data. The moments of individual digitisations were studied to see if the moments could be used to calculate the drift time of, and hence characterise, the digitisations. The best representation of the digitisations in terms of the moments achieved an average compression of 81.5 percent. However resolution of the drift time calculated using the moments was poorer by a factor of 1.44 than when calculated by standard QT analysis.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Chapter 1

Introduction

The volume of digital data in the world is currently increasing at a unprecedented rate. In all areas of computer use the number of bytes of data that need to be stored and transmitted is growing at an enormous rate, giving rise to problems to the users and maintainers of the data. For example word processing programs that occupied only a few tens of kilobytes in the early 1980s can easily occupy as much as 10 megabytes today. High energy physics experiments also suffer from these problems, the data collection rate of the ATLAS [1] experiment will be two orders of magnitude higher than the currently running H1 [2] experiment.

Due to this vast increase in the amounts of data, storage devices that appeared gigantic only a few years ago have suddenly become inadequate. More importantly, the sheer volume of data being sent down networks requires that the capacity of the networks be increased, to carry all the traffic users demand, with the increases in cost that implies.

One, partial, solution to these problems is data compression. Within theoretical limits compression routines introduce computational complexity to reduce communications bandwidths and storage requirements.

1.1 Data Compression In Physics

High energy physics experiments use analogue to digital converters (ADCs) to convert the analogue signals from detector sub units such as calorimeters and drift chambers into digital information. The digital data is then moved around the Data Acquisition (DAQ) system of a detector until it is either stored or discarded.

The ADCs used in current detector experiments operate at speeds of around 100MHz, i.e. one digital sample, or time slice, of the analogue signal is taken approximately once every 10 nanoseconds. The ADCs are termed Flash ADCs (FADCs) because of this high speed of data collection. The resolution of these FADCs is typically 8 to 12 bits, with some systems using a non linear conversion of the analogue signal to a digital value to increase the effective digitising range.

A typical FADC memory stores 256 time slices. Digitising each time slice with a resolution of 8 bits requires a FADC memory of 256 bytes. Individually, each 256 byte digitisation is a small amount of data, however tens of thousands of digitisation are recorded for each event in current detectors. Future detectors such as the ATLAS detector may well have tens of millions of digitisations per event. The data production rates of some typical past, present and future detectors are shown in table 1.1. In addition to the increase in the number of digitisations per event, future detectors will operate at higher event rates and will have have higher particle multiplicities, i.e. more tracks per event.

The data from the FADCs is passed up the DAQ system by data buses. Each data bus has a finite bandwidth which limits the speed at which the data can be moved around the system. The amount of bandwidth required is therefore determined by the amount of data transferred for each event and the event rate. The larger the bandwidth of a data bus the greater the cost and the cost of connecting the modules of the DAQ system. For the H1 experiment the DAQ system formed approximately 25 percent of the total cost of a detector.

Reducing the large amounts of data generated by detector experiments will,

	UA1	H1	ATLAS - CMS
Online	1983	1992	2004
Tracking	10^4 channels	10^4 channels	$10^6 - 10^8$ channels
Calorimetry	10^4 channels	5×10^4 channels	10^5 channels
Muons	10^4 channels	2×10^4 channels	10^6 channels
Raw Data	10^2 gigabytes/sec	3×10^4 gigabytes/sec	10^6 gigabytes/sec
Level 1	0.1 gigabytes/sec	0.1 gigabytes/sec	10^3 gigabytes/sec
Level 2	0.01 gigabytes/sec	0.02 gigabytes/sec	10^2 gigabytes/sec
Level 3	0.01 gigabytes/sec	0.005 gigabytes/sec	1 gigabyte/sec
Tape Write	1 megabyte/sec	1 megabyte/sec	100 megabytes/sec
Event Size	100 kilobytes	125 kilobytes	1 megabyte

Table 1.1: Data production rates of selected experiments

therefore, reduce the cost of building the detector. In addition the size of the data that is permanently stored, and hence the cost of storing the data, will be reduced.

The aim of this study was to design and test compression methods for use with data from FADCs by adapting existing compression techniques and developing new compression methods. The methods described later are software based, although the goal of the study is to find compression methods that could be feasibly be transferred to silicon.

1.2 Types Of Compression

The methods of data compression can be separated into two types: lossless and lossy compression methods. Lossless compression methods store all of the original data in a more compact form so that the original data can be restored. As lossless compression methods are required to store all of the original data compression ratios (a measure of the amount of compression, see section 3) are typically 1.5 to 8, depending on the redundancy in the information and the complexity of the compression algorithm. Lossless methods are required, for example, for computer programs and documents.

Lossy compression methods store only part of the input data or extract the important information from the data, so that the original data can not be restored. The degree of acceptable data loss varies greatly, depending on the application. For example, several compression routines for handling graphics files drop information to which the eye is not very sensitive, e.g. subtle variations in colour saturation. Compression ratios for lossy compression methods are much greater than for lossless compression, ratios of 100 or more can be achieved.

Chapter 2

The H1 Data Acquisition System

2.1 The H1 Detector

The data used to compare the compression methods given in chapters 3 and 4 came from the H1 detector at the HERA accelerator. The accelerator collides 28 GeV electrons with 821 GeV protons. This leads to an asymmetric design for the H1 detector [3], as shown in figure 2.1.

Specifically, the data used came from the radial modules of the forward track detector. The forward track detector consists of 3 identical supermodules. Each supermodule consists of 3 planar drift chambers which are orientated at 0^o, $+60^o$, -60^o to the vertical, followed by a multi wire proportional chamber, a transition radiator and finally a radial drift chamber.

The radial chambers each consist of 48 wedge shaped segments. Each wedge contains 12 sense wires which are separated by 1 cm along the beam direction. The sense wires are alternately staggered by 288μm out of the true radial wire plane, this is so that the data from the radial chambers can be used to determine which side of the wire plane a particle passed. Each of the sense wires is connected to a partner 105^o around the module. The wire pairs are read out at the circumference of the

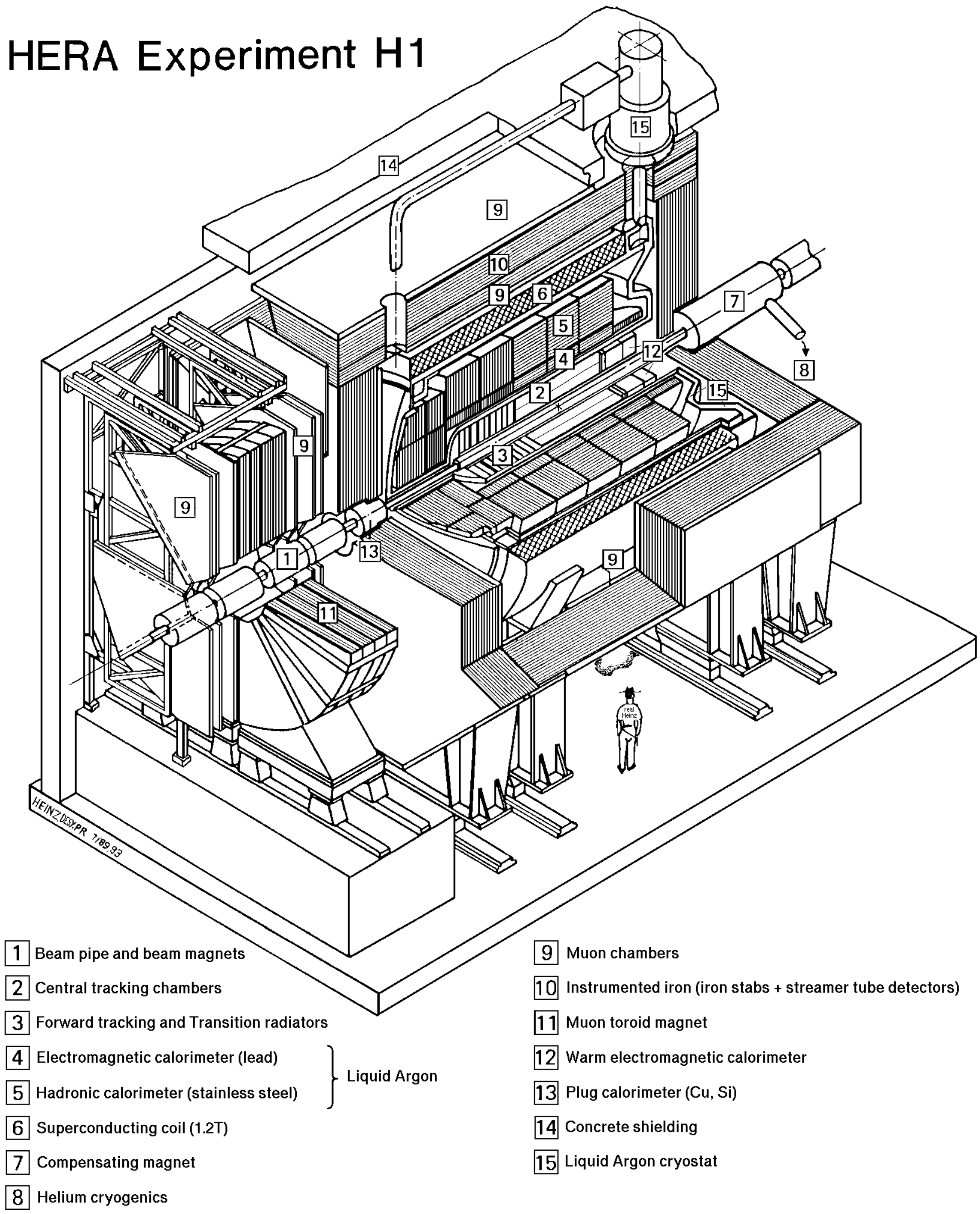

Figure 2.1: Schematic of the H1 detector

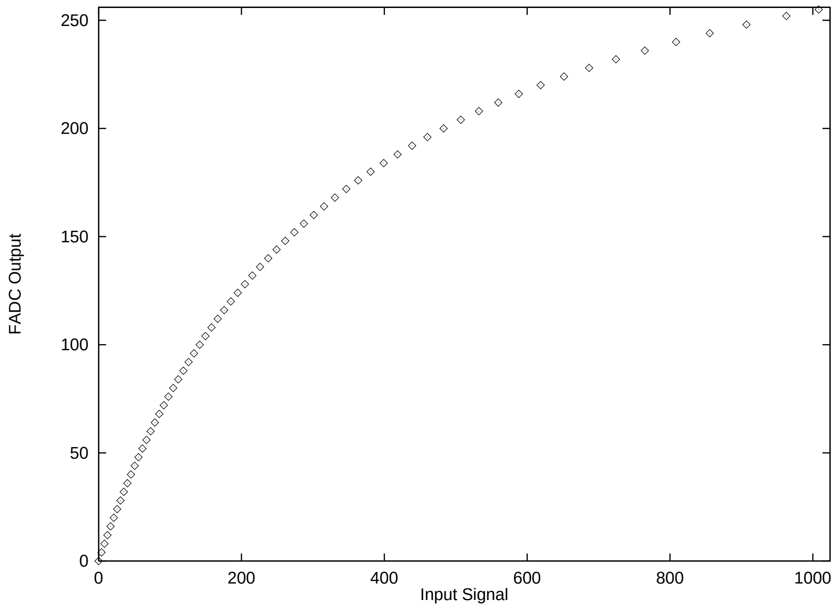

Figure 2.2: The non linear response of the FADC

radial module. For lossless compression the arrangement of the radial chambers is unimportant, but for lossy compression the arrangement was used to form a method of evaluating the compression methods, see section 4.2.1.

2.2 The FADCs

The analogue signals from the wire ends of the tracking chambers (both the radial and planar chambers) are quantised to a digital signal by FADCs, operating at 104MHz. This corresponds to a digitisation every 9.6 nanoseconds. The FADCs used in this experiment have an 8 bit resolution and have a non linear response, in that they are less sensitive to large signals, as shown in figure 2.2. The non linear response allows for a greater range of digitisation without increasing the bit resolution (and hence the cost) of the FADCs.

Each time slice in the digitisation from the FADCs is re-linearised before data analysis using equation 2.1, where u is the unlinearised value of the time slice from the FADC data and l is the linearised value for the time slice.

$$l = \frac{1024 * u}{1024 - u * (1024 - 256)/256} \tag{2.1}$$

2.3 The Scanners

The scanners are devices which are activated by a fast trigger signal and then search for hits in the digitisations from the FADCs [4]. There are three scanners for each of the three radial modules. When a scanner is triggered, the FADCs are first stopped and then the FADC memories are copied into the internal memory of the scanner. The memory to memory transfer takes place at 80 megabytes/sec. Once all of the FADC memories have been copied to the scanners the FADCs are re-enabled for the next event.

Once the FADC digitisations have been copied into the scanner's memory, the memory is scanned for hits. A hit is found if two successive time slices are greater than a preset start threshold. The end of the hit is marked by two successive time slices below a second, end, threshold. Both the start and end thresholds can be adjusted in the control software. A table which contains the positions of the start and end of each hit found, as well as the FADC number from which the data came, is built up in the memory of the scanner.

2.4 The Front End Processors

If the event is not rejected, then the Front End Processors (FEPs) [5] are triggered. The time slices between the start and end points calculated by the scanner only contain the part of the hit which is above the thresholds, this excludes both the start and end of the hit. To correct this the FEPs copy data from a preset number of time slices before the start point to a preset number of time slices after the end point. This data is copied into the FEPs memory and assembled into formatted data banks.

At this point, depending on the detector, the FEPs can pass the data down the DAQ pipe line or perform on-line QT analysis on the data to calculate the drift times and charge integrals. The method used for calculating the drift time and the charge integral is optimised for each detector system.

All the data banks are then assembled by a master FEP into a full data bank which is the starting point for off line analysis.

2.5 Radial QT Analysis

The radial data in the data banks from the FEPs is first linearised using equation 2.1 and then searched for hits using a weighted difference of samples. A hit is found when two or more time slices are above a set threshold in the weighted difference of samples, possibly with intervening time slices below threshold. The end of the hit is found when at least two time slices in the weighted difference of samples are negative, without the weighted difference of samples of any intervening time slices being above the threshold [6].

The difference of samples (DOS) of the data from a particular FADC is defined as:

$$DOS_i(n) = FADC_i(n) - FADC_i(n-1) \tag{2.2}$$

where $FADC(n)$ is the linearised value of the nth time slice either from end $i = 1$ or 2. The weighted difference of samples, $W(n)$, is defined as:

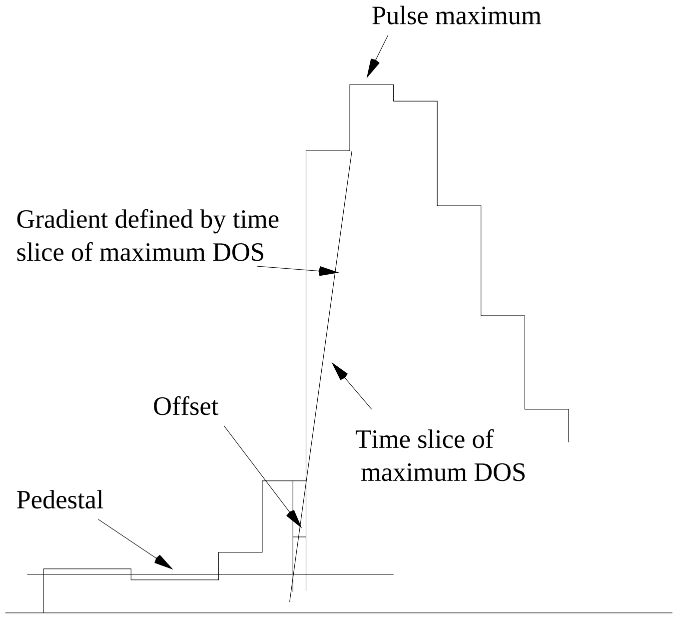

Figure 2.3: Time determination for the radial data

$$W(n) = \sum_{i=1,2} (FADC_i(n) - P_i - A) \times \sum_{i=1,2} DOS_i(n) \tag{2.3}$$

where P_i is the pedestal level (the average of the flat region of data at the beginning of the sample) and A is an arbitrary offset from the pedestal to allow for the tuning of the detection efficiency for small hits.

The time of the of hit is calculated in two parts. The first estimate of the time is the time slice, n, with the largest value of $DOS_1(n) + DOS_2(n)$, in the data above the threshold in $W(n)$.

The times for both ends of the wire are then calculated separately, by finding the time slice with the largest difference of samples within two time slices of the first estimate. This value is refined by projecting the gradient given by the difference of samples of this time slice back down to the pedestal level as shown in figure 2.3. This offset is then corrected on the assumption that the distribution of offset values should be a constant over the width of a time slice and zero else where.

The time values for both wire ends are then averaged, weighted by the charge integral of the digitisation. The charge integral is calculated for each wire end by summing the time slice height above the pedestal level for 12 time slices starting 2 time slices before the calculated hit time and using linear interpolation within the time slices at each end of the range.

Multiple hits are analysed by calculating the time and charges of the first hit, restricting the charge integration if necessary to finish before the start of the next hit. The tail of the first hit is then removed from the data by using the value of the time slice immediately before the next hit and using this to subtract off the tail of the earlier pulse, assuming an exponential signal decay. The next hit is then analysed as before.

2.6 Planar QT Analysis

As with the radial data the planar data is first linearised. The digitisations are then scanned for regions where adjacent time slices exceed the pedestal value of the digitisation (the average of the first five time slices) plus a fixed value. If at least four adjacent time slices exceed this value then the region is identified as a cluster.

Each cluster found is then searched for hits. The difference of samples is calculated and hits are identified as groups of at least two adjacent time slices in the difference of samples that are greater than zero and that the sum of the difference of samples of the group is above a fixed threshold. The time of each hit is calculated as the weighted average over the time slices of the group:

$$t = \frac{\sum\limits_{group} n \cdot DOS(n)}{\sum\limits_{group} DOS(n)} \tag{2.4}$$

Multiple hits within a cluster can be separated if at least one time slice in the difference of samples is negative.

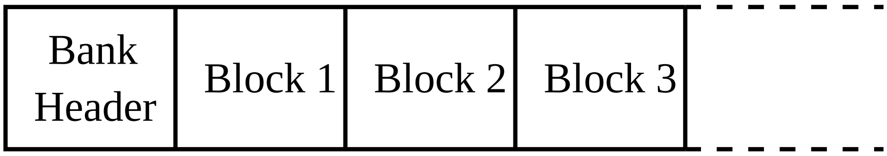

Figure 2.4: Structure of a data bank

2.7 Data Format

As the data is moved up the DAQ system it is packed into ever larger and larger data structures. The largest is the data bank which contains the stored data of one type for a single event. The data bank consists of header followed by data blocks (figure 2.4).

The logical structure of a data bank header is shown in figure 2.5. Each data bank header consists of four, four byte words. The first word is a four letter name which identifies the type of data in the data bank. The NUMBER word holds the event number from which the data in the data bank came from. FLAGS contains status information about the bank which is used by the DAQ system. Finally LENGTH gives the total length of the data area containing the data blocks.

The data banks are transferred around the DAQ system using direct memory access. For the transfer to be efficient the banks are stored at the start of 16

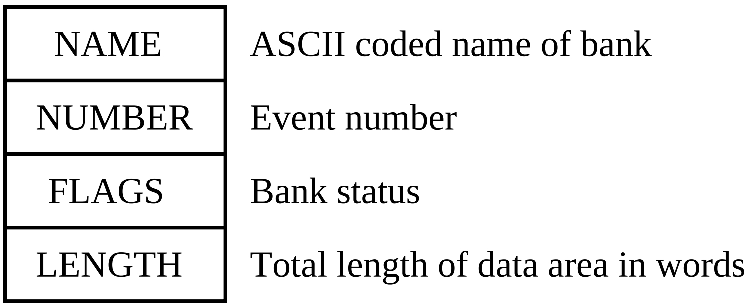

Figure 2.5: Structure of a data bank header

Figure 2.6: Structure of a data block

byte words and are padded out so that their length is a multiple of 16 bytes.

The data blocks contain many types of data, e.g. digitisations from the FADCs and times calculated by QT analysis from systems throughout the detector. The format of the data block is similar to the data bank. The data block consists of a header followed by a data area, see figure 2.6.

The header consists of four 16 bit words (figure 2.7). NWORD is the total number of 16 bit words in the block, including the header. IFLAGS contains the type of data in the data area and some status information. For data blocks containing digitisations from either the radial or planar chambers then the final two words are IDFADC which is the channel number of the FADC from which the digitisations came, and ISTART which is the position in the FADC memory from which the first digitisation was taken. The data blocks are constrained to start on 4 byte boundaries and are padded out to be a multiple of 4 bytes in length.

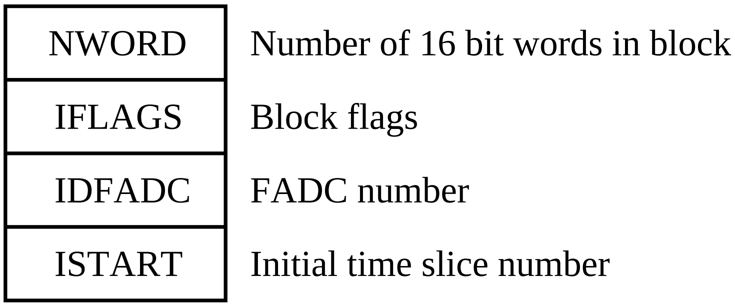

Figure 2.7: Structure of a data block header

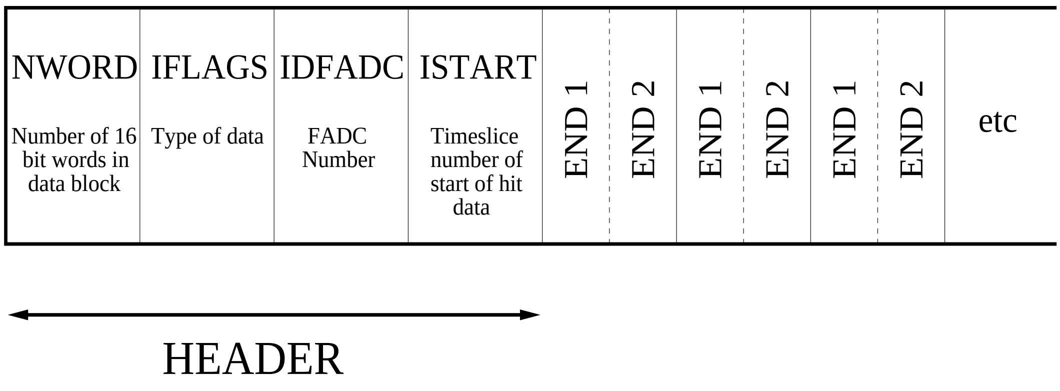

Figure 2.8: Format of a radial data block

Each radial data block stores the data from both ends of one wire. The format is shown in figure 2.8. If an even number of time slices are stored then the data block will be a multiple of 4 bytes in length. However an odd number of time slices requires two pad bytes to be added, either at the end of the data or between ISTART and the 1st digitisation. With the presence and position of any pad bytes indicated by the value of IFLAGS.

2.8 The Raw Data

The data on which the compression algorithms were developed and tested came from the forward radial chambers of the H1 detector. The data consisted of 149,776 digitisations from 918 events, with digitisations containing both single and multiple hits.

Figure 2.9 shows a linearised digitisation with one hit. The data from both wire ends is shown, which in this case is a large signal from one end and a smaller signal from the other end. The difference of samples for this digitisation is shown in figure 2.10.

A multi hit digitisation is shown in figure 2.11. The difference of samples

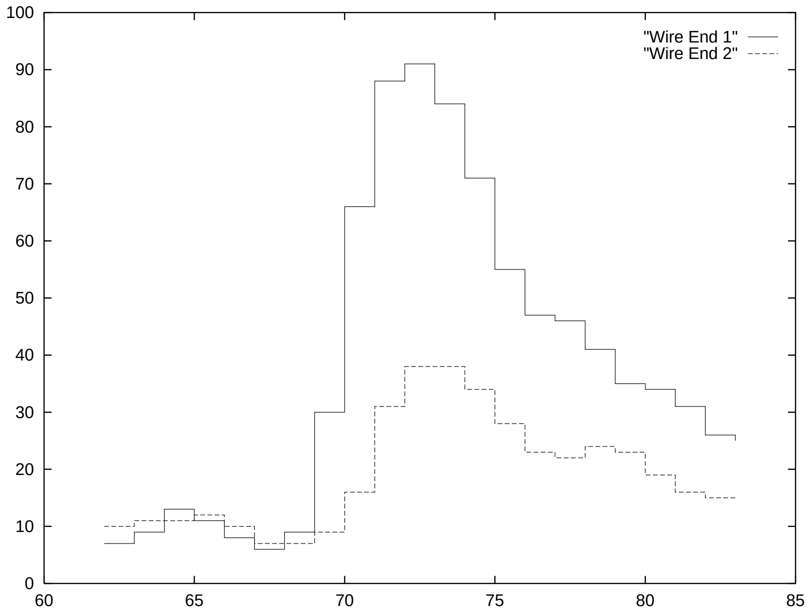

Figure 2.9: Digitisation showing only one hit

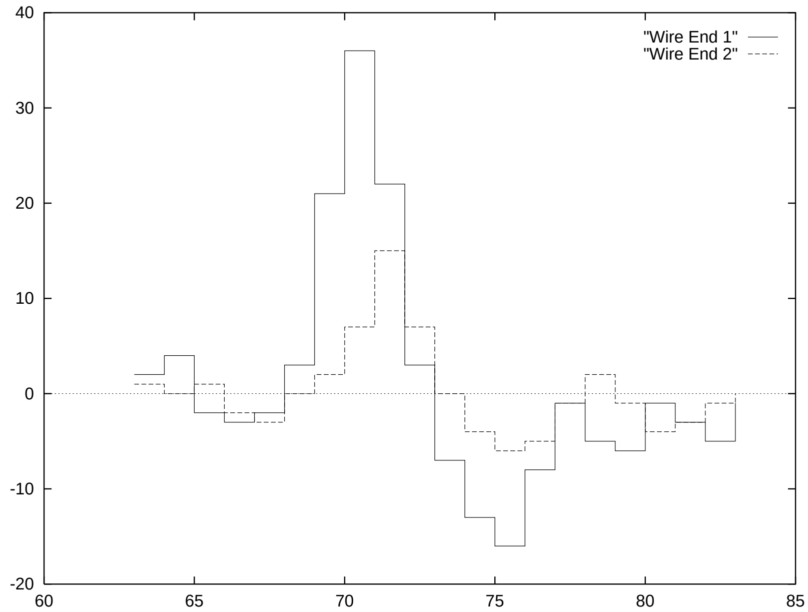

Figure 2.10: Difference of samples of digitisation showing only one hit

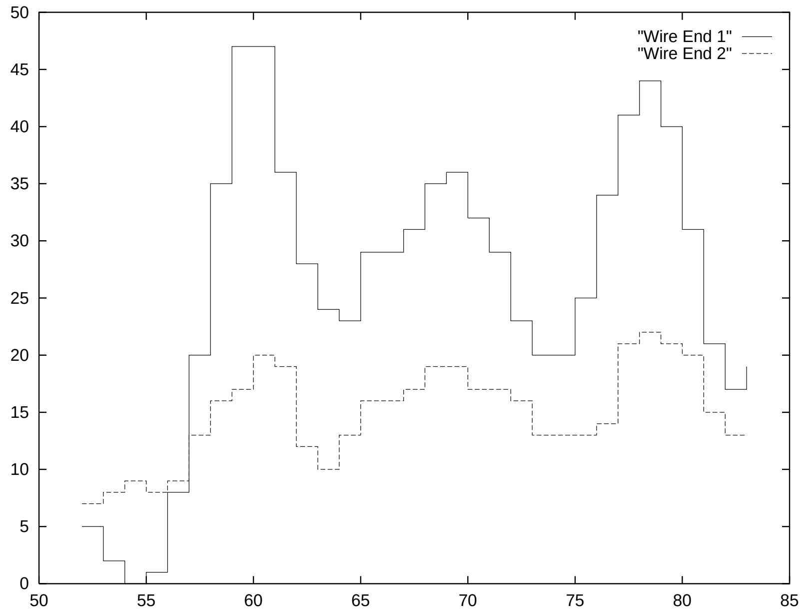

Figure 2.11: Digitisation showing multiple hits

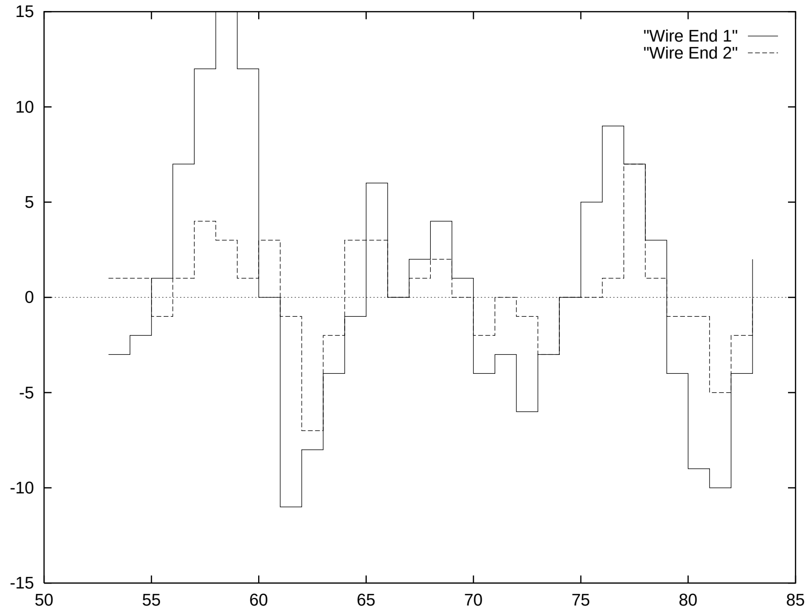

Figure 2.12: Difference of samples of digitisation showing multiple hits

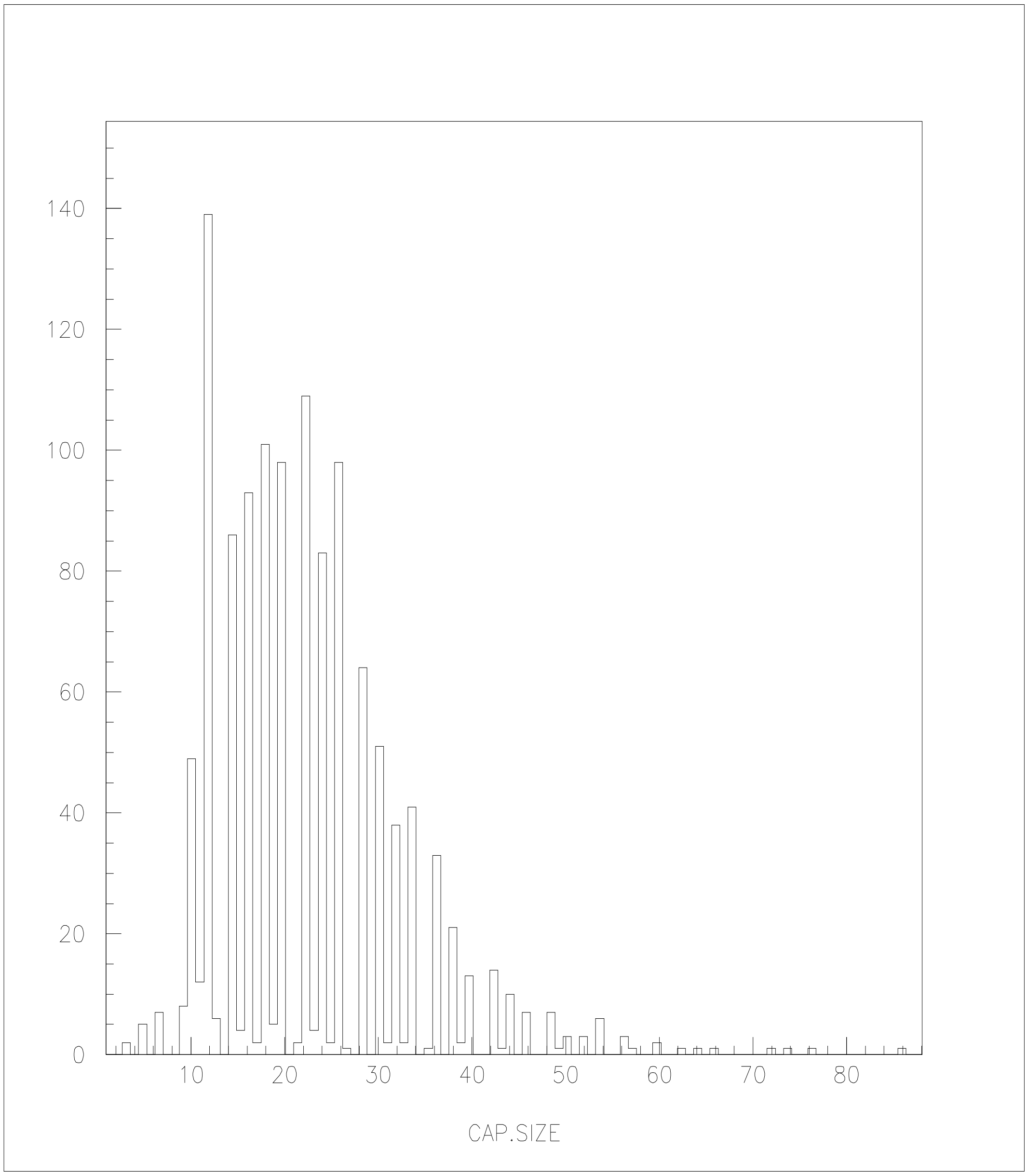

Figure 2.13: Distribution of digitisation lengths

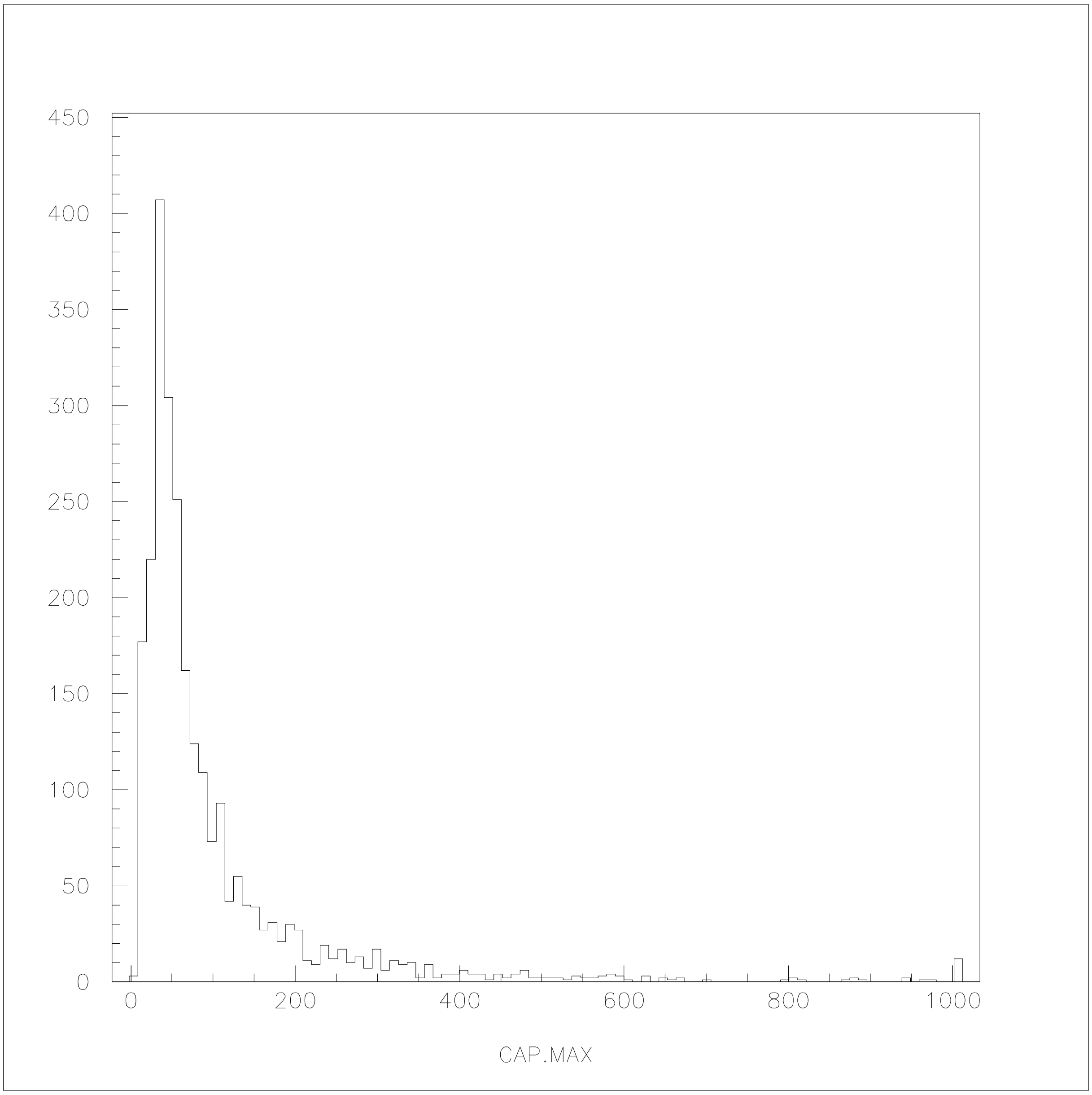

Figure 2.14: Distribution of digitisation heights

for this digitisation is shown in figure 2.12. Digitisations with lengths of 50 time slices or more are mostly multiple hit digitisations.

The distributions of the length and height of the digitisations in the data file used are shown in figures 2.13 and 2.14. The digitisation lengths are mostly an even number of time slices in length. This is due to packing in the data into blocks where odd length digitisations would have to to padded out wasting space which could be used to store data. The odd length digitisations occur only in special cases, e.g. if in the original 256 byte digitisation two adjacent hits are found and the second hit finishes at the end of the 256 byte digitisation this second digitisation could not be expanded to be a even number of bytes in length.

2.9 Computer Code

The data file used for this analysis contained the data from the forward radial chambers for 918 events. The individual digitisations were stored in a format called BOS [7], which is a data format developed at DESY. The data for each event are stored in a separate BOS bank. The BOS banks are collected together in a file known as an F-record. These records are created by a library of routines called FPACK [8], which is a set of stand alone Fortran and C programs for machine independent input and output of data. The FPACK and BOS library routines allow the data in FPACK files to be accessed without knowing the details of the format of the data files.

The compression programs were written in the C computer language. C was chosen for the ease at which data manipulation at the bit and byte levels can be performed. The BOS and FPACK library functions are called using Fortran calling conventions. The differences in calling conventions between C and Fortran can be handled by coding the C calls to the routines appropriately, however such coding is, in general, machine dependent. A set of machine independent macros which make it easier to call Fortran functions from C, have been written at CERN and they are called cfortran.h [9]. The definitions of the macros used in the file are changed depending on the machine used to compile the C program, but the C source code does not need to be changed when compiling on different machines.

Chapter 3

Lossless Methods

Lossless compression is the process of reducing the size of data but still allowing the original data to be reproduced from the compressed data. This chapter describes and compares the methods of least number of bits, difference of samples with the least number of bits, Huffman encoding of both the digitisations and difference of samples of the digitisations, Arithmetic encoding again of both the raw digitisations and difference of samples of the digitisations. In addition the results of compressing the whole data file using generally available compression programs are shown.

3.1 Terminology

A standard measure of the degree of compression is the compression ratio, defined in equations 3.1 and 3.2. The uncorrected compression ratio, R_U, is simply the ratio of the uncompressed data size to the compressed data size. The corrected compression ration, R, is the ratio of the uncompressed data size and the sum of the compressed data size, the size of any header required to interpret the compressed data and the size of any padding required by the compression algorithm and/or storage medium.

$$R_U = \frac{uncompressed\ size}{compressed\ size} \tag{3.1}$$

$$R = \frac{uncompressed\ size}{compressed\ size\ +\ header\ size\ +\ padding} \tag{3.2}$$

A second measure of data compression is the percentage compression, which is:

$$P_U = \frac{uncompressed\ size\ -\ compressed\ size}{uncompressed\ size} \times 100 \tag{3.3}$$

or for a corrected percentage compression:

$$P = \frac{uncompressed\ size\ -\ (compressed\ size\ +\ header\ +\ padding)}{uncompressed\ size} \times 100 \tag{3.4}$$

The compressed data was stored in the same way for each compression method. Each digitisation was compressed and stored separately. The data header was first stored followed by the compressed data for the digitisations from each wire end and then any pad bits to pad out of the length of the data to an integer number of bytes.

3.2 Limits For Lossless Compression

The limits for lossless compression were first determined by Shannon [11]. The extent by which a message may be losslessly compressed is limited by the message's entropy. In this sense entropy is a measure of the message's information content: the more probable the message, the lower it's entropy. Expressed in bits, the entropy is minus the logarithm to base two of the message's probability (equation 3.5).

$$entropy = -\,log_2(probability) \tag{3.5}$$

The probability and hence entropy of a given symbol in a message depends on the model used to analyse the message. For example suppose the probability of any given letter in a message written in English being a u is 0.01. This has an entropy of 6.6 bits, i.e. it requires 6.6 bits to encode. However using a more advanced model that uses the previously encoded letter to guess the next then after a q the probability of a u rises, for example, to 0.95 which can be encoded in only 0.074 bits. Coding to fractional bits is difficult and time consuming which is why most coding schemes lose compression by only encoding to whole bits. Arithmetic compression, described latter, will encode to fractional bits.

The maximum percentage compression expressed as a function of the average entropy per symbol, in bits is:

$$max\ compression = \frac{average\ no\ of\ bits\ per\ symbol\ -\ average\ entropy}{average\ no\ of\ bits\ per\ symbol} \times 100 \tag{3.6}$$

3.3 Initial Data Reduction

Some reduction of the data already takes place in the detector system. The non linear response of the FADCs effectively compresses a 10 bit digitisation range into 8 bits. Although some accuracy is lost from large digitisations, this does not effect the QT analysis to any great extent. When the FEPs copy data from the scanners only data containing hits are copied. This is a massive reduction in data size. Typically for the radial chambers only 200 digitisations out of 864 are copied and for these only, typically, 30 bytes of the original 256 byte FADC memory is copied to the DAQ pipe line.

The combination of not storing FADC memories which do not contain any hits and then only storing the parts of the FADC memories which contain hit data reduce the data moved up the DAQ system by a factor of about 30. Thus the data compression techniques described in this section and the following section start with this reduced, unlinearised data as the uncompressed data.

3.4 Compression Using The Unlinearised Raw Data

3.4.1 Compression Limits

In this section the unlinearised data is compressed directly. The data is treated as a memoryless source, i.e. the data is treated as if there is no relationship between the value of one time slice and the next. For a memory less source the average entropy of a single time slice can be expressed as:

$$entropy = -\sum_{i=0}^{255} p_i log_2(p_i) \tag{3.7}$$

where p_i is the probability of any individual time slice having the value i in any digitisation.

The entropy of the unlinearised test digitisations using memoryless encoding schemes was 6.37. Therefore the maximum compression using a memoryless source model was:

$$max\ compression = \frac{8 - 6.37}{8} \times 100 = 20.375\% \tag{3.8}$$

3.4.2 Bit Reduction

From the distribution of the maximum digitisation heights, figure 2.14, it can be seen that most digitisations do not require the full eight bits to encode. Bit reduction compresses the data by storing the time slices from each digitisation in the minimum number of bits required to store the largest time slice.

The compression header requires 14 bits, 8 bits to store the number of time slices in each digitisation and two of lots 3 bits to store the size of each compressed time slice in bits of each wire end, to allow the compressed data to be decompressed.

Although the digitisation could in principle need between 0 and 8 bits to store each time slice as each digitisation has some none zero time slices the number of bits required to store the largest time slice is between 1 to 8 bits. This can be stored in 3 bits as the number of bits to store largest time slice minus one.

Using the bit reduction method gives a compression ratio of 1.13, or a reduction of 11.5%.

3.4.3 Sectioned Bit Reduction

From the shapes of the digitisations, figures 2.9 and 2.11, each hit in a digitisation starts of at a low level, the pedestal, peaks sharply and then falls of back to the pedestal level and possibly peaking again for digitisations containing multiple hits. Thus the number of bits required to store a time slice is initially low at the start of the digitisation, then becomes large at the peak and then falls away. Using the Bit Reduction method to store each time slice in the number of bits needed to store the largest time slice is therefore wasteful for the majority of the digitisation. By sectioning the digitisation and storing each section in the number of bits required for the largest time slice in that section, an improvement on the Bit Reduction method should be gained.

The smallest section possible would be an individual time slice. While storing each time slice in the smallest number of bits would efficiently store the time slice data itself, each time slice would then require a header of 3 bits to store the number of bits the time slice was stored in. The 3 bit header per time slice would cause this method to actually increase the data size.

For maximum compression using the sectioning method the individual section should be neither to large so that compression is inefficient as when the digitisation is treated as a single section in the first bit reduction method or to small so that number of section headers required cancels any compression gained. The algorithm used to section the digitisation was:

Step 1: First create one section for each of the time slices in the digitisation and calculate the number of bits required to store each time slice.

Step 2: Concatenate together adjacent sections that require the same number of bits to store a time slice.

Step 3: Now calculate the header size required for each section by calculating the number of bits need to hold the size of the largest section minus one, as no section will have a length of zero. Then calculate the number of bits required to store the number of bits minus one needed to store the largest time slice in each section.

Step 4: Concatenate any adjacent sections where the number of bits required to store the two sections separately is greater then the number of bits required to store the two sections as one section with the bits per time slice the larger of the bits per time slice of the two separate sections.

Step 5: Swap any time slices from the end of one section to the beginning of another, of visa versa if the time slice requires the same or less bits than the other section and the other section has a smaller bits per time slice than the current section the time slice is in.

Step 6: Recalculate the number of bits required to store the header data of the new sections. This will only change the no of bits required to store the length of each section.

Step 4 in the algorithm is difficult to code efficiently. The method used was:

Substep 4.1: Initialise k to be the number of time slices in the digitisation.

Substep 4.2: Loop through the sections of the digitisations and if the current section length equals k and the next section has a length greater than zero and combining the two sections takes less space than storing the two sections separately then concatenate together the two sections.

Substep 4.3: subtract one from k

Substep 4.4: repeat steps 2 and 3 until k equals zero.

Sectioned bit reduction gave a compression ratio of 1.19, for a percentage reduction of 16.0%.

The extra compression gained over the Bit Reduction method is small when compared to the extra complexity introduced.

3.4.4 Huffman Encoding

The Huffman compression technique is named after the person who first proposed it in 1952 [10]. This compression technique is used in modern FAX machines.

FAX machines operate by scanning and digitising an image to be transmitted. The resolution of this digitisation is 3.85 or 7.7 lines per millimetre vertically and 8.05 pixels per millimetre horizontally. For a typical scanned page this gives approximately two million binary digits. Transmitting this at 4800 bits per second would require in excess of six minutes.

In most documents the majority of scanned lines consists of long runs of white pixels whilst other lines comprise a mix of long runs of white and long runs of black pixels.

Instead of transmitting only runs of only binary 1s or 0s the length of each run of white or black pixels is transmitted. The code words which represent the

Run Length	Code Word For White Pixels	Code Word For Black Pixels
0	00110101	0000110111
1	000111	010
2	0111	11
3	1000	10
4	1011	011
5	1100	0011
6	1110	0010
7	1111	00011
8	10011	000101
9	10100	000100

Figure 3.1: CCITT facsimile conversion codes

length of each pixel sequence are a Huffman code. The code is fixed in that as part of the process to create the transmission protocol extensive analyses of scanned document pages were made. From this the frequency of the length of runs of white and black pixels were measured and a Huffman code was calculated separately for both white and black pixel runs.

As a side note the data is sent as a length of pixel runs only and a technique known as overscanning is used which means that all lines start with at least one white pixel. Table 3.1 gives part of the Huffman code used in FAX machines. So the code 10011,0010,10100,10 (the commas are included for ease of human interpretation of the code and the FAX machine receives the code as 1001100101010010) indicates a run of 8 white pixels followed by 6 black pixels, then 9 white pixels and then 3 black pixels. Uncompressed the message requires 26 bits and compressed requires 16 bits, for a compression ratio of 1.625. In normal use the lengths of runs of similar pixels is usually much higher and a compression ratio of 10 is typical.

3.4.5 Construction Of Huffman Codes

A Huffman encoding scheme attempts to construct variable length codes of binary 1s and 0s for each of the input symbols. With short length codes to

Symbol	Probability
A	0.25
B	0.2
C	0.2
D	0.18
E	0.09
F	0.05
G	0.02
H	0.01

Figure 3.2: Example Symbol Probabilities

represent common symbols and longer codes to represent less common symbols, so that, overall, compression is achieved.

The construction of the codes is complicated in that no information about the length of a code symbol is stored prior to the code symbol its self. Each code symbol, therefore, can not start any other longer code symbol. For example if a symbol is represented by the code 1110 then 1, 11 and 111, cannot be used to represent other symbols. However 1111 would be a valid code as it would be distinguishable from 1110.

To construct a Huffman code the probability of each symbol occurring in the message needs to known before encoding begins so that the code symbols can be determined. The method to construct a Huffman code, in outline is:

Step 1: Generate a list of the probability of occurrence of each of the source symbols.

Step 2: Take the two smallest probabilities in the list and connect them together to form the parent branch point. Label the branch from the parent to one of child probabilities 1 and the other 0.

Step 3: Replace the two probabilities in the list with a single probability which is the sum of the two probabilities. If only one element remains in the list then the tree is complete other wise repeat steps 2 and 3.

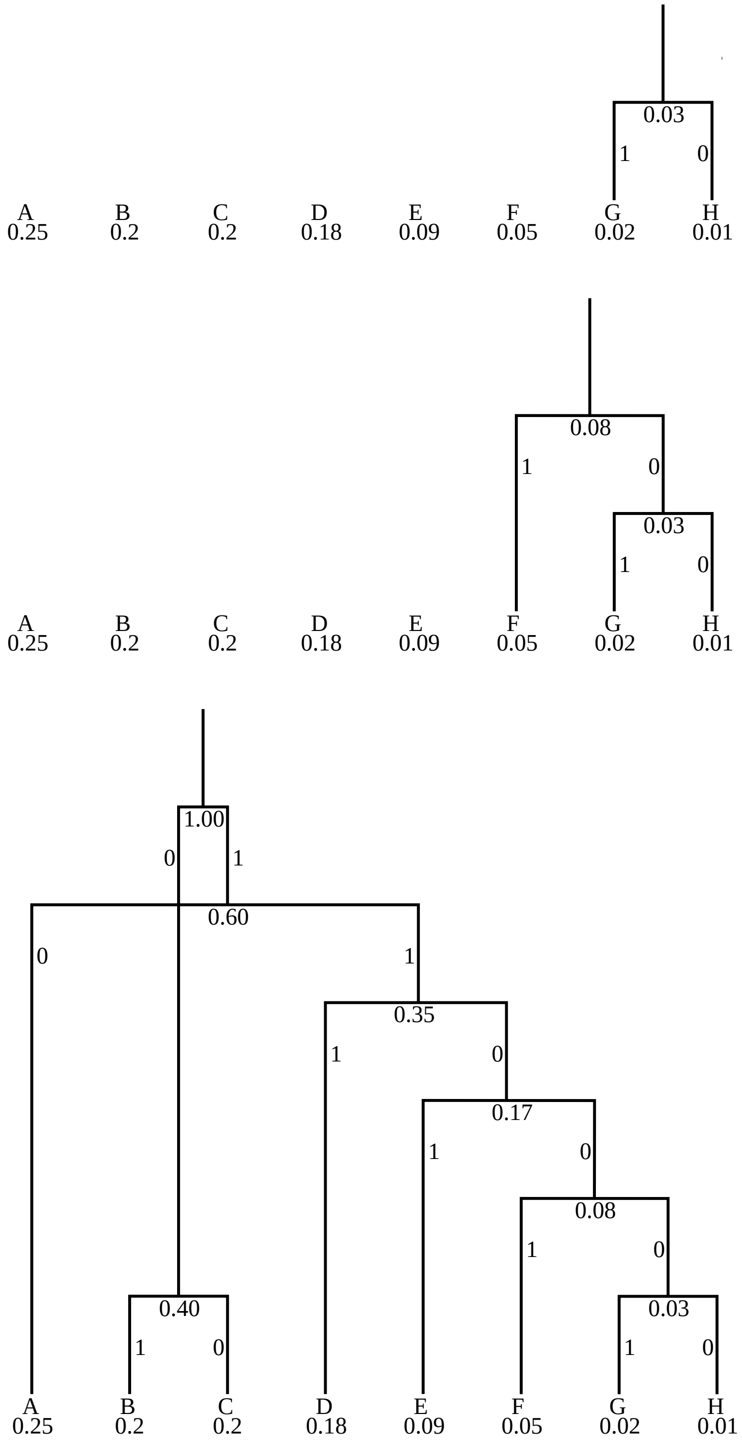

Figure 3.3: Construction of a Huffman tree

For example using the letters A to H as the symbols and with the probabilities shown in figure 3.2. The two least probable letters are combined and their probabilities added to find the probability of the combination. In the example G and H are combined to give a combined probability of 0.03, figure 3.3 top. Then the next two lowest probabilities are combined, i.e. the F and the G/H pair.

The procedure is repeated until all all of the letters have been combined into a Huffman tree. When a branch is made, or after the tree has been completed each branch of the tree is labelled with a 0 or a 1. How the 0 and 1s are assigned is not important, in the example the 0 is assigned to the branches with the lower probability.

To generate the Huffman code for a given letter the tree is traversed from the top of the tree to the letter. Thus, in the example, the letter A is coded as 10, D as 111 and G 110001. The most likely letters are on the shortest branches and are encoded with the fewest bits, whilst the least likely letters are on the longest branches and require the most bits.

This method of generating codes ensures that code symbols will not be confused during decompression. The prefix of one symbol will never be a unique symbol in itself. In the example the code for D is 111, the method of constructing the Huffman tree ensures that 1 and 11 are not used to encode other letters.

Huffman encoding of the digitisations gives a compression ratio of 1.22, i.e. a compression percentage of 18.0, close the maximum compression of 20% for memoryless unlinearised data.

3.4.6 Arithmetic Compression

With the previous techniques a whole number of bits were required to encode each input symbol. If an input symbol's entropy indicates that only, for example, 0.5 bits are required to encode the symbol then the previous methods described will not achieve optimal compression.

Arithmetic coding [12] can code to fractional bits. While arithmetic codes are more complex to implement, current work [13] has made practicable implementation possible.

Arithmetic coding compresses a message by merging the probabilities of all the symbols in the message into a single high-precision fraction which is the compressed message. The fraction is constructed such that it can be decoded unambiguously. Similarly to Huffman encoding the probabilities of the various symbols in a message need to be determined before encoding. Each symbol is then assigned a probability range in the interval [0,1) [1]. While the positions of the various ranges are arbitrary, they must not overlap, and the widths are equal to the probabilities of the symbols they represent. The fraction is then constructed by:

Step 1: Begin with a current interval [L,H), initialised to [0,1).

Step 2: Subdivide the current interval into subintervals, one for each of the possible input symbols. The size of the subinterval is proportional to the probability of the symbol.

Step 3: Select the subinterval corresponding to the input symbol and make this the current interval.

Step 4: Output sufficient bits to distinguish the current interval from all other intervals.

At first sight the algorithm looks impractical as digital systems have limited precision and will overflow after a few symbols. However the encoded fraction need not be stored until the end of the input message and then transmitted. As soon as the most significant digits in the upper and lower boundaries become equal, they are guaranteed not to change as additional symbols are encoded. Thus they may be transmitted and the remaining fraction can be expanded to fill the full range of the variable storing the fraction.

If the message is of variable length then to indicate the end of the message an end of message symbol can be added to the message alphabet.

[1] A [or] bracket indicates an inclusive limit and a (or) bracket an exclusive limit

Symbol	Probability	Range
a	0.1	[0, 0.1)
e	0.1	[0.1, 0.2)
g	0.2	[0.2, 0.4)
i	0.2	[0.4, 0.6)
l	0.1	[0.6, 0.7)
m	0.1	[0.7, 0.8)
p	0.1	[0.8, 0.9)
r	0.1	[0.9, 1.0)

Letter	Range	Interval width	Interval after symbol encoding	Output
initially			0.0 - 1.0	-
p	0.8 - 0.9	1.0	0.8 - 0.9	-
i	0.4 - 0.6	0.1	0.84 - 0.86	8
l	0.6 - 0.7	0.02	0.852 - 0.854	5
g	0.2 - 0.4	0.002	0.8524 - 0.8528	2
r	0.9 - 1.0	0.0004	0.85276 - 0.85280	-
i	0.4 - 0.6	0.00004	0.852776 0.852784	7
m	0.7 - 0.8	0.000008	0.8527816 - 0.8527824	8
a	0.0 - 0.1	0.0000008	0.8527816 - 0.85278168	1 6
g	0.2 - 0.4	0.00000008	0.852781616 - 0.852781632	-
e	0.1 - 0.2	0.000000016	0.852781617 - 0.852781619	1

Figure 3.4: Example of arithmetic encoding using decimal fractions

Figure 3.4 shows an example of arithmetic encoding. In using arithmetic coding to encode the word *pilgrimage* the component letters are first assigned probabilities derived from the model of the data, although in this example the probabilities come from the message word itself. Then each symbol is assigned a unique range in the interval [0,1) whose width is equal to the letter's probability. The process of compressing the message consists of progressively narrowing the arithmetic interval as each symbol is added. Initially the interval is [0,1). After the first letter is encoded it is [0.8,0.9), then [0.84,0.86) and so on. The arithmetic compression routines used were written by A. C. Popat [14], unlike the rest of the compression software which was written by the author.

The compression ratio for arithmetic compression was 1.01 which corre-

	Individual Digitisation	Individual Wire	Single Event
Bit Reduction	11.5	12.6	13.5
Sectioned Bit Reduction	16.0	17.0	18.0
Huffman Encoding	18.0	19.0	19.9
Arithmetic Encoding	1.5	10.9	20.2

Table 3.1: Compression percentages

sponds to a compression reduction of 1.5%, a surprisingly poor result. This was the result of encoding each digitisation separately, as the arithmetic compression routine used outputs the whole of the fraction remaining after the was input symbol in the message has been encoded. This increases the size of the compressed message, all most removing the compression completely.

3.5 Size Of Compressed Data

Compression routines are more efficient if used on a single large data file than on several small data files if the compressed data must be padded to be an integral number of bytes in length. In addition arithmetic compression is very inefficient when used on small data files.

Table 3.1 shows the compression percentages for the different compression methods described so far on three sizes of data, compressing individual digitisations, compressing the two digitisations from a single wire and compressing the digitisations from the radial chambers for a single event.

3.6 Lossless Compression Using The Difference Of Samples

The difference of samples is the difference between the value of one time slice and the next. The reason for using the difference of samples is to reduce the

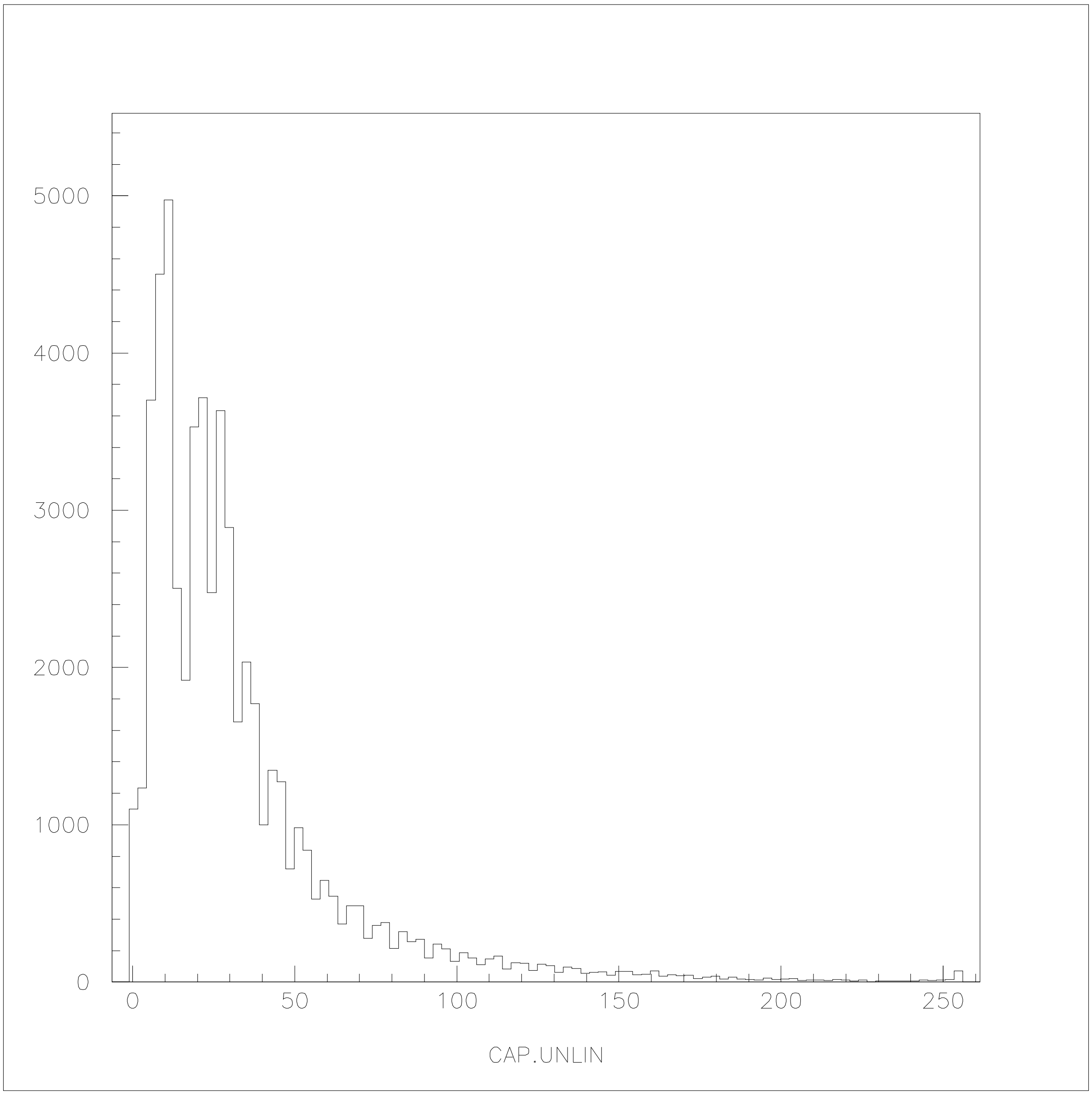

Figure 3.5: Distribution of unlinearised time slice values

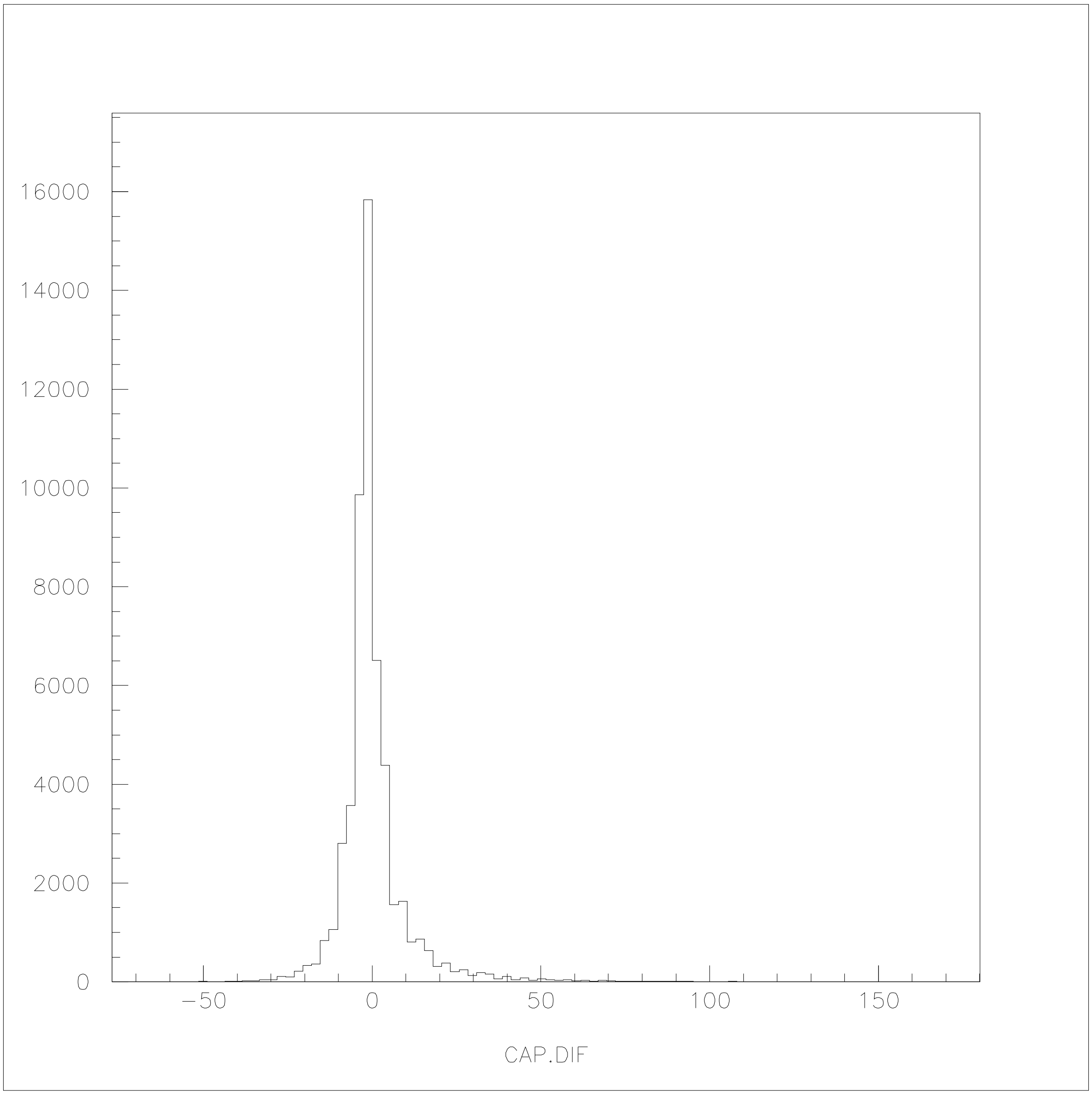

Figure 3.6: Distribution of difference of samples values

	Individual Digitisation	Individual Wire	Single Event
Bit Reduction	16.1	17.2	18.2
Sectioned Bit Reduction	24.4	25.4	26.3
Huffman Encoding	35.3	36.2	37.2
Arithmetic Encoding	19.3	28.8	38.3

Table 3.2: Compression percentages for DOS

range of the most common values used to store the digitisation. Figure 3.5 shows the distribution of the values of the unlinearised time slices, while figure 3.6 shows the distribution of the difference of samples. Although the range of possible values increases from 0 to 255 to -255 to 255, the variance of distribution, compared to the unlinearised time slice distribution has been reduced from 33.5 to 10.9.

As with compressing the unlinearised digitisations the methods used are still memoryless and the average entropy is again calculated using equation 3.7

The entropy of the unlinearised difference of samples test digitisations using memoryless encoding schemes was 4.86. Therefore the maximum compression using any difference of samples method was:

$$max\ compression = \frac{8 - 4.86}{8} \times 100 = 39.25\% \tag{3.9}$$

In order to restore data stored as a set of differences from the previous time slice the absolute value of the first time slice needs to be stored as part of the compression header of the unlinearised data. This adds 8 bits to the size of the compression header. The previously used compression methods were then used on the unlinearised difference of samples.

The results for different data sizes are given in table 3.2. Arithmetic compression again suffers if the data is compressed on a digitisation by digitisation basis. Both the Huffman and Arithmetic compression methods, when used on the radial data on an event by event basis approach the limit for compression for a memoryless treatment of the difference of samples. For compressing digitisations separately, Huffman compression would clearly be the best choice.

	Size	% Compression
Original file	8728200	-
Bit Reduction	7020000	19.6
Sectioned Bit Reduction	6973200	20.1
Huffman Encoding	6903000	20.9
DOS Bit Reduction	6645600	23.9
DOS Sectioned Bit Reduction	6388200	26.8
DOS Huffman Coding	5639400	35.4
compact	6960256	20.3
compress	7365723	15.6
gzip	6534933	25.1
zip	6534920	25.1
DOS Bit Reduction + zip	5917541	32.2
DOS Sectioned Bit Reduction + zip	6093993	30.2
DOS Huffman Coding + zip	5409160	38.0

Table 3.3: Comparison with generally available compression programs

3.7 Large Scale Compression

The primary aim for these compression methods is to be used as close to the FADCs as possible and to be integrated into hardware on a detector. As secondary option the compression methods could be used to compress the event data that is eventually stored on tape. This would be done by compressing the data from one type of sub detector (e.g. the forward radial chambers) in a single BOS bank.

Table 3.3 shows the results of compressing a data file from the forward radial chambers with several of the compression methods given here and *compact*, *compress*, *gzip*, *zip* which are generally available general compression programs. The data file consisted of 918 events, each event having a BOS header bank (HEAD), and a forward radial raw data bank (FRRD). Huffman compression is again clearly the best choice and is better than the generally available compression programs.

Chapter 4

Lossy Compression

The techniques of lossy compression all involve identifying the important information in the given data and storing only that information. An example of lossy compression currently used in the H1 detector occurs when the 256 byte digitisations are scanned for hits and only the approximately 30 byte blocks containing hits are passed on up the DAQ system.

4.1 Lossy Compression Of Planar Chamber Data

A simple way of designing a lossy compression routine is to examine which parts of the data are actually used later and then to store only this part of the data.

The data sent from the scanners attached to the planer chambers is searched for hits by first linearising the digitisations and then forming the difference of samples of the digitisations. The rising edges of pulses are sharp positive peaks in the difference of samples. A hit is found if a cluster of at least two adjacent time slices in the difference of samples satisfy two conditions: that each time slice must have a positive difference of samples and that the sum of all the time slice difference of samples exceed a fixed threshold. The time of the hit is then determined using equa-

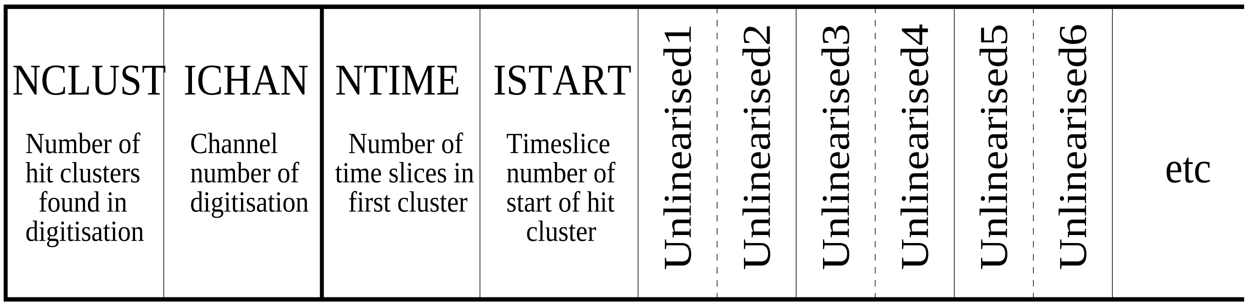

Figure 4.1: Cluster data block

tion 2.4 which uses only the linearised value of the time slice difference of samples of the cluster.

4.1.1 Clusters

Instead of storing each digitisation, compression can be achieved by storing only the clusters in the difference of samples of the digitisations that satisfy the hit finding rules for the planar chambers given above.

First the unlinearised difference of samples of the digitisation is formed. The difference of samples is then scanned through to find the first time slice with a positive difference of samples. The difference of samples is scanned from this point to find the position of the first time slice with a zero or negative difference of samples value. The two time slices then define the position of the cluster, the first time slice being an inclusive limit the second an exclusive limit. If the cluster is less than two time slices in length it is dropped at this point. If longer the value of the difference of samples for each time slice in the cluster is summed.

If this is value exceeds a set threshold then the cluster is stored in the format shown in figure 4.1. Each block would store all the clusters found in one digitisation. For detectors which have data from both ends of a sense wire, then the data from both wire ends would be stored if any one wire ends data was identified as a cluster. In the figure NCLUST is the number of clusters found in the digitisation, ICHAN is the number of the FADC to identify the origin of the data. These two values form the header of the data block and would be encoded in 3 bytes. Four bits for the number of clusters, for a range of 0 to 15, 16 bits for ICHAN to give 65,536 possible separate channels, with the remaining 4 bits set aside as flag bits, to be used if required. Each cluster recorded in the data block would start with a byte (NTIME) to record the number of time slices in the cluster and one byte (ISTART) to record the number of the time slice of the first difference of samples value in the cluster. The unlinearised value of the difference of samples of each time slice in the cluster would then be stored, one byte per time slice.

This method achieved a compression ratio of 3.30 or a percentage reduction of 69.7% over all. This includes digitisations where no clusters were found and stored.

4.1.2 Lossy Huffman Encoding Of The DOS

Huffman encoding of the difference of samples of digitisations when used to losslessly compress a digitisation achieves a compression reduction of approximately 35 percent. When calculating the drift time using standard QT analysis from planar chamber data, only the values of the difference of samples of time slices with positive difference of samples values are used, although not every time slice with a positive difference of samples value is used. The value of time slices with a difference of samples value that is zero or negative are not used, only the fact that the time slice difference of sample is zero or negative is used. This can be used to increase the compression ratio achieved by Huffman compression of the difference of samples (see previous chapter) by setting all negative difference of samples values to zero, see figure 4.2.

The compression achieved is similar to the previous cluster compression, with a compression ratio of 3.03 or a percentage reduction of 67.0%.

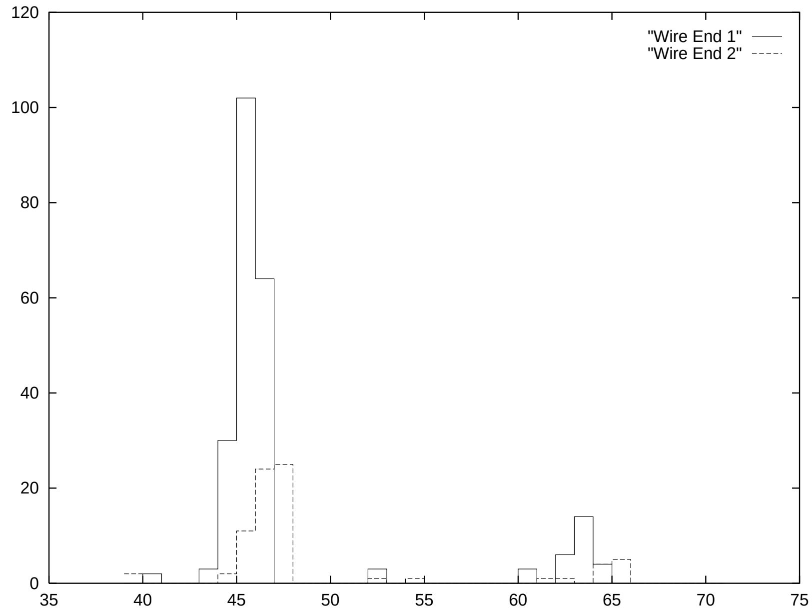

Figure 4.2: DOS of a digitisation with all negative DOS values set to zero

4.2 Moments

The previous two lossy compression methods reduced the amount of data by storing only the information required by the QT analysis method. The aim of this method is to use only a few numbers to characterise a digitisation, and from these numbers calculate the drift time. The numbers used here are the moments of the digitisation. Moments are used because any function can be described by its moments. For example Gaussian distributions require only two moments to characterise the distribution, the mean and the variance. While a skewed Gaussian requires an additional moment, the skew.

The 1st moment of the digitisation is defined as:

$$m_1 = \frac{1}{\sum x_i} \sum_{i=1}^{N} x_i i \tag{4.1}$$

where i is the number of the time slice and x_i is the linearised value of the

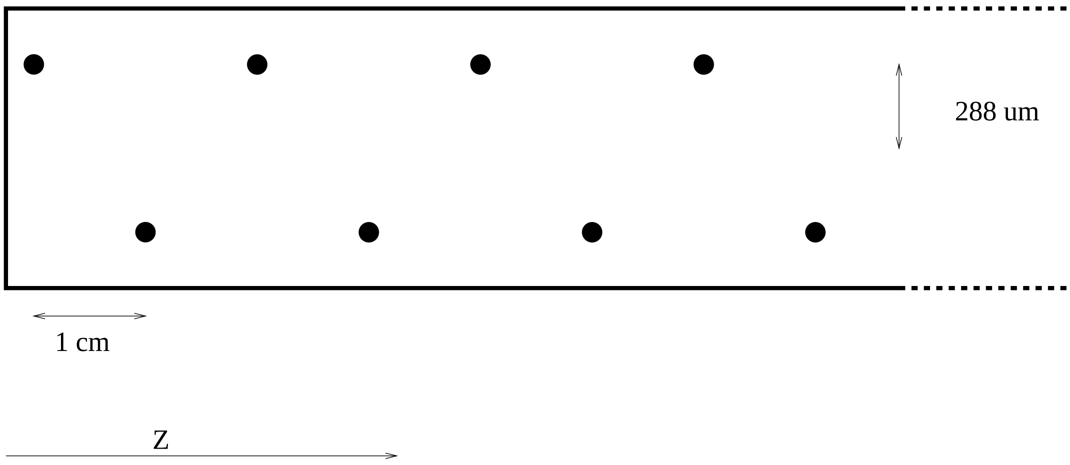

Figure 4.3: Schematic of a radial wedge chamber

i^{th} time slice. The subsequent moments are then defined as:

$$m_n = \frac{1}{\sum x_i} \sum_{i=1}^{N} \left((i - m_1)^n \right) x_i \tag{4.2}$$

The drift time, t, is then expressed as a function of these moments. The required function is calculated by comparing the drift time calculated using standard QT analysis and the moments of the digitisation.

$$t = f(m_1, m_2, ..._{,n}) \tag{4.3}$$

4.2.1 Radial Chambers

In order to evaluate if moments can be used to calculate drift times then some knowledge of the physical layout of the radial chambers is required.

The forward radial tracking chambers are arranged into 3 groups of 48 wedges. Each group of 48 wedges is arranged in a disc around the beam pipe. Each wedge contains 12 sense wires each lying radially outwards from the beam pipe and arranged into a plane along the beam pipe (figure 4.3). The sense wires are staggered alternately 288μm either side of the centre of the wire plane.

The signals from the wires are read out from both wire ends. This is achieved by connecting each sense wire to the corresponding sense wire 105° around the radial module. This allows the read out electronics to be placed only on the outside of the radial chamber modules.

To determine the accuracy of using moments to calculate the drift times as opposed to the standard QT analysis methods the error on the drift times returned by the various methods needs to be calculated.

One way of calculating this error is to fit the times returned for the wires in a single wedge to a straight line, and then to calculate the difference between the fit and the drift time. From these differences the average error on the drift time would then be calculated. This method has some problems in that the stagger of the wires must be corrected for and if the particle track crosses the wire plane the drift times would no longer fit to a straight line. The wire stagger can be corrected for either by adding/subtracting a constant value from the drift time or by fitting the data to two straight lines, one for wires staggered one way and the other for wires staggered the opposite way.

A better method is to use the drift times from 3 adjacent wires to form a checksum value. The drift times returned are related to the distances:

$$x_0 + d = v(t_0 + T_0) \tag{4.4}$$

$$x_1 - d = v(t_1 + T_1) \tag{4.5}$$

$$x_2 + d = v(t_2 + T_2) \tag{4.6}$$

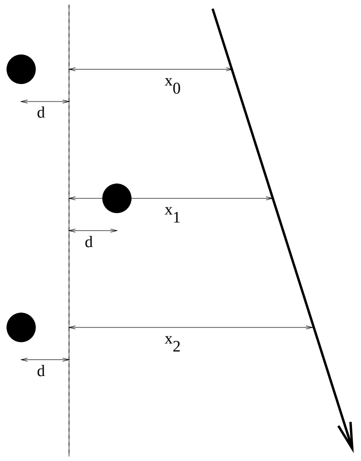

Figure 4.4: Checksum calculation

where x_0, x_1 and x_2 are the drift distances to the centre of the wire plane; d is the wire stagger distance; t_0, t_1 and t_2 are the drift times; T_0, T_1 and T_2 are timing constants which in the case of the radial chambers are all equal and v is the drift velocity which is assumed to be a constant.

Assuming the particle track is straight, i.e. the particle has a high momentum, then the drift distance x_1 is the average of x_0 and x_2.

$$x_1 = \frac{x_0 + x_2}{2} \tag{4.7}$$

rewriting this as:

$$x_0 - 2x_1 + x_2 = 0 \tag{4.8}$$

and substituting the drift times in and setting $T_0 = T_1 = T_2 = T$ gives:

$$v(t_0 - T) - d - 2v(t_1 - T) - 2d + v(t_2 - T) - d = 0 \tag{4.9}$$

which simplifies to:

$$t_0 - 2t_1 + t_2 = \frac{4d}{v} \tag{4.10}$$

the value of which is called a checksum. Plotting the checksum values as a histogram gives a histogram with two peaks one centred on $+4d/v$ the other centred on $-4d/v$. The negative peak is produced by particles that have passed on the other side of the wire plane, where the sign of d in equations 4.4, 4.5 and 4.6, is reversed.

Particles that pass through the wire plane will give invalid values for the checksum. The checksum values for each wire in the wedge should alternate in sign as the sense of the wire stagger changes. This method has the advantage that the wire stagger is eliminated without the need for any corrections.

The error in the drift times can be obtained from the histogram of the checksum values using the propagation of errors:

$$\sigma^2_{checksum} = \sigma^2_{t_0} + \sigma^2_{t_1} + \sigma^2_{t_2} \tag{4.11}$$

assuming each drift time has the same error then:

$$\sigma_t = \frac{\sigma_{checksum}}{\sqrt{6}} \tag{4.12}$$

where $\sigma_{checksum}$ is the average width of the peaks in the checksum histogram.

4.2.2 Compression Limits

The amount of storage space required for any representation of a digitisation using moments, depends on two parameters: the number of moments required

No. Moments	2 bytes/moment		4 bytes/moment	
	Comp. gain	% Reduc.	Comp. gain	% Reduc.
1	10.8	90.7	5.4	81.5
2	5.4	81.5	2.7	63.0
3	3.6	72.2	1.8	44.4
4	2.7	63.0	1.35	25.9

Table 4.1: Compression limits for moments

to represent the data and the accuracy to which the moments need to be stored, the more accuracy required, the more bytes required to store the values of individual moments.

Table 4.1 shows the compression gain and percentage reduction in data size obtained for representations requiring up to four moments and for each moment requiring 2 or 4 bytes to store.

4.2.3 Checksum Calculation And Data Selection

The same radial data used for the lossless compression analysis was used for the moment analysis. In order to compare the moments and the QT times the selection of digitisations to be used from the data set was critically important. The criteria used were:

1. Clean single hits. Any wires with more than one digitisation were dropped. As were digitisations which returned more than one hit from QT analysis.

2. A large number of checksums per wedge (6 or more). In order to calculate if the particle track through a wedge crossed the wire plane several checksums are required.

3. Consistent checksum values within a wedge. If the checksums within a wedge were found to be inconsistent the data from the wedge was dropped.

Figure 4.5 shows the checksum histogram obtained for the QT times. Mea-

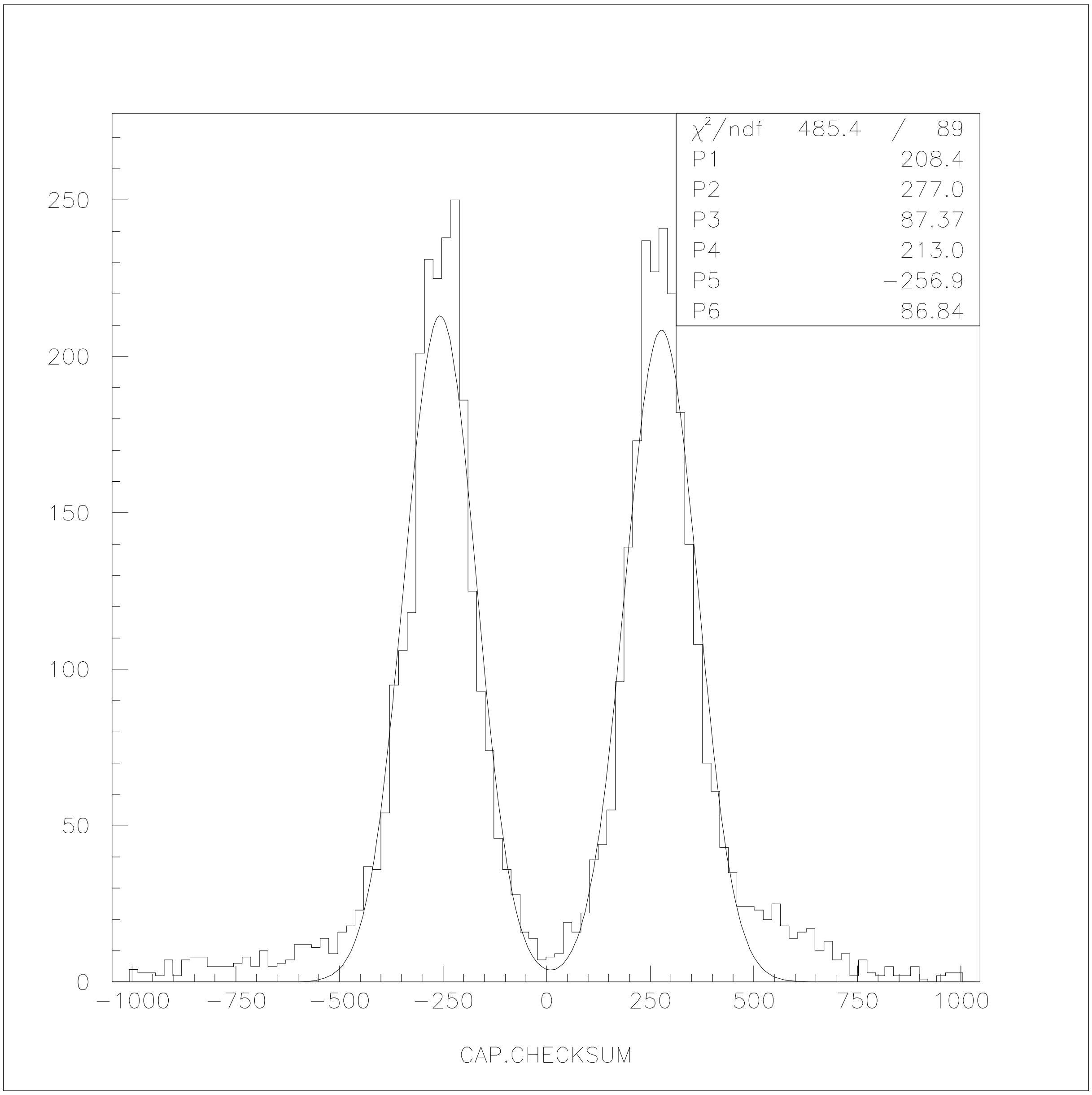

Figure 4.5: QT checksum

Moment	Correlation
m_1	0.996
m_2	0.032
m_3	-0.017
m_4	0.022

Table 4.2: Correlations of moments with QT drift time

suring the full width at half maximum and assuming the central part of the peak approximates to a Gaussian distribution gives an average peak width of 68 which is equivalent to a time error of 5.3ns or a position uncertainty of 265μm.

4.2.4 QT times, Moment Correlation

The drift time used to correlate the moments can be obtained in several ways. A fit to the drift times of the wires in a wedge can be used to give a better estimate of the true drift time than the single drift time returned from QT analysis, although the individual QT drift times can be used.

In addition the drift time returned by QT analysis is the average of the two drift times obtained from the digitisations from either end of a single sense wire. The individual drift times for each wire end can be recovered and used to correlate with the moments calculated from each wire end.

For this analysis of moments, the individual drift times for each wire end were used to correlate with the moments calculated from each wire end's digitisation.

The first moment of the digitisation is highly correlated with the drift time as can be seen in figure 4.6. This is to be expected since the 1st moment is the centre of gravity of the digitisation which is a good, but not perfect estimation of the drift time. After the first moment the correlation between the drift time and the higher moments is near zero. A correlation plot of the drift time with the second moment is shown in figure 4.7. Table 4.2 summarises the correlation of each of the first four moments with the drift time.

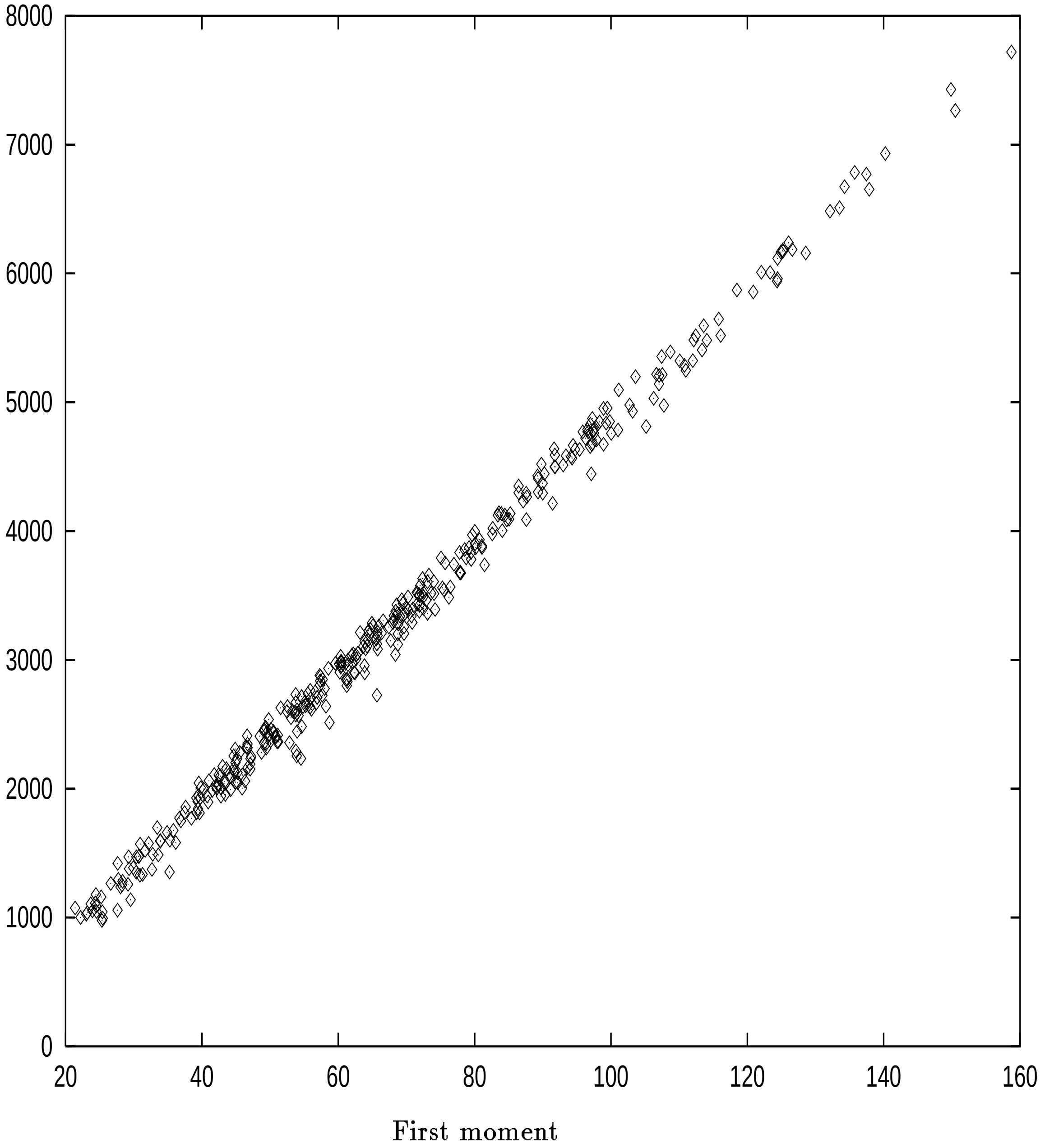

Figure 4.6: First moment plotted against QT drift time

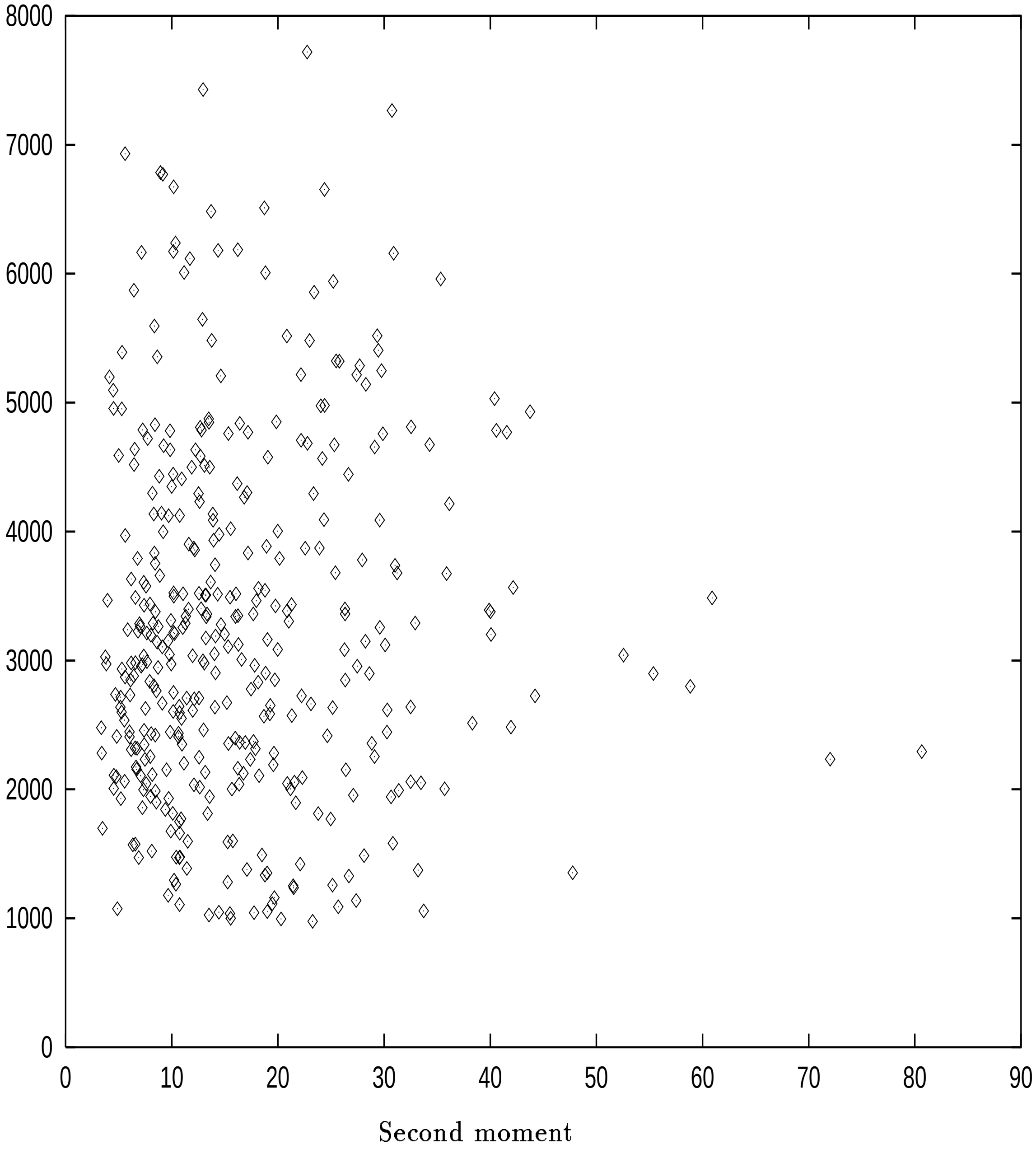

Figure 4.7: Second moment plotted against QT drift time

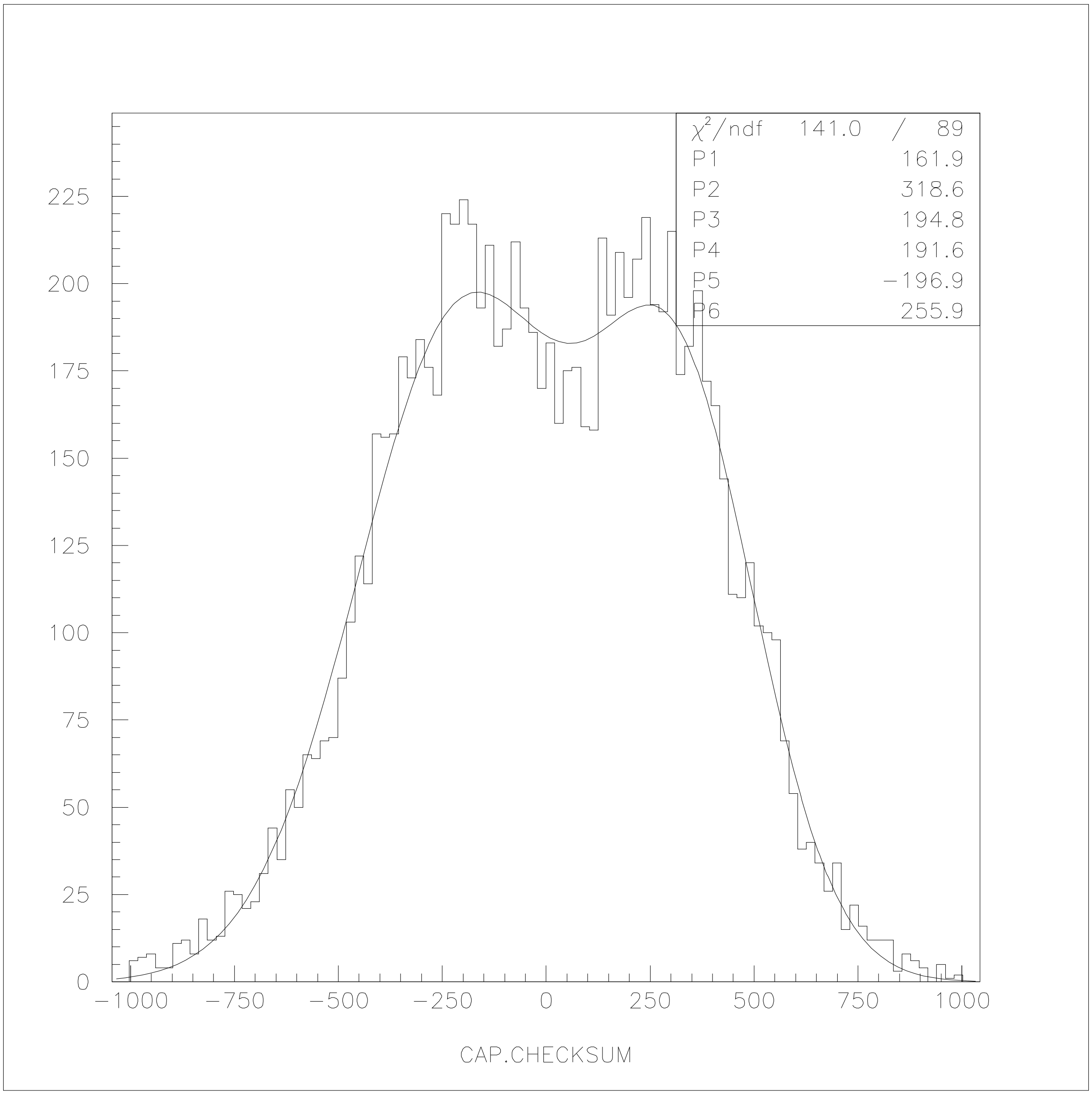

Figure 4.8: Checksum calculated using first moment

Fitting the QT drift times and the first moment to a straight line gives the first method of calculating the drift time from the first moment. Although the correlation between the QT drift time and the first moment is very high, just using the first moment to calculate the drift time gives a result which is significantly poorer than when using standard QT analysis to calculate the drift time. Figure 4.8 shows the checksum calculated using drift times calculated from the first moment. The two separate peaks in the QT checksum have very nearly merged into a single peak.

The higher moments have too low a correlation with the drift time to be used directly to improve the calculation of the drift time from the first moment.

Figure 4.9 shows the correlation of the second moment with the residuals of the fit of the first moment to the QT drift time. The second moment is clearly correlated to the residuals although in a nonlinear manner. The plot of second moment against the residuals was fitted to a polynomial of order x^5 and used to correct the drift time calculated using the first moment. The correlation coefficient for this plot was -0.743.

A plot of the checksum histogram calculated from the first two moments (figure 4.10) clearly shows two peaks, although of a lower resolution than the checksum peaks calculated using QT drift times.

The next logical step world be to try to use the third moment to improve the drift time resolution. Figures 4.11 and 4.12 show the correlation of the third moment to the QT drift time and to the residuals of the drift time calculated from the first and second moments. As can be seen from the graphs no correlation is present. Similar plots using the fourth moment were also done, and again no correlation was found. This leaves no clear continuation for using moments.

Although the resolution of drift times calculated using the first two moments is worse than the resolution of drift time calculated using standard QT analysis, if this poorer resolution is acceptable then the amount of compression achieved by using the two moments to represent the digitisation needs to be determined.

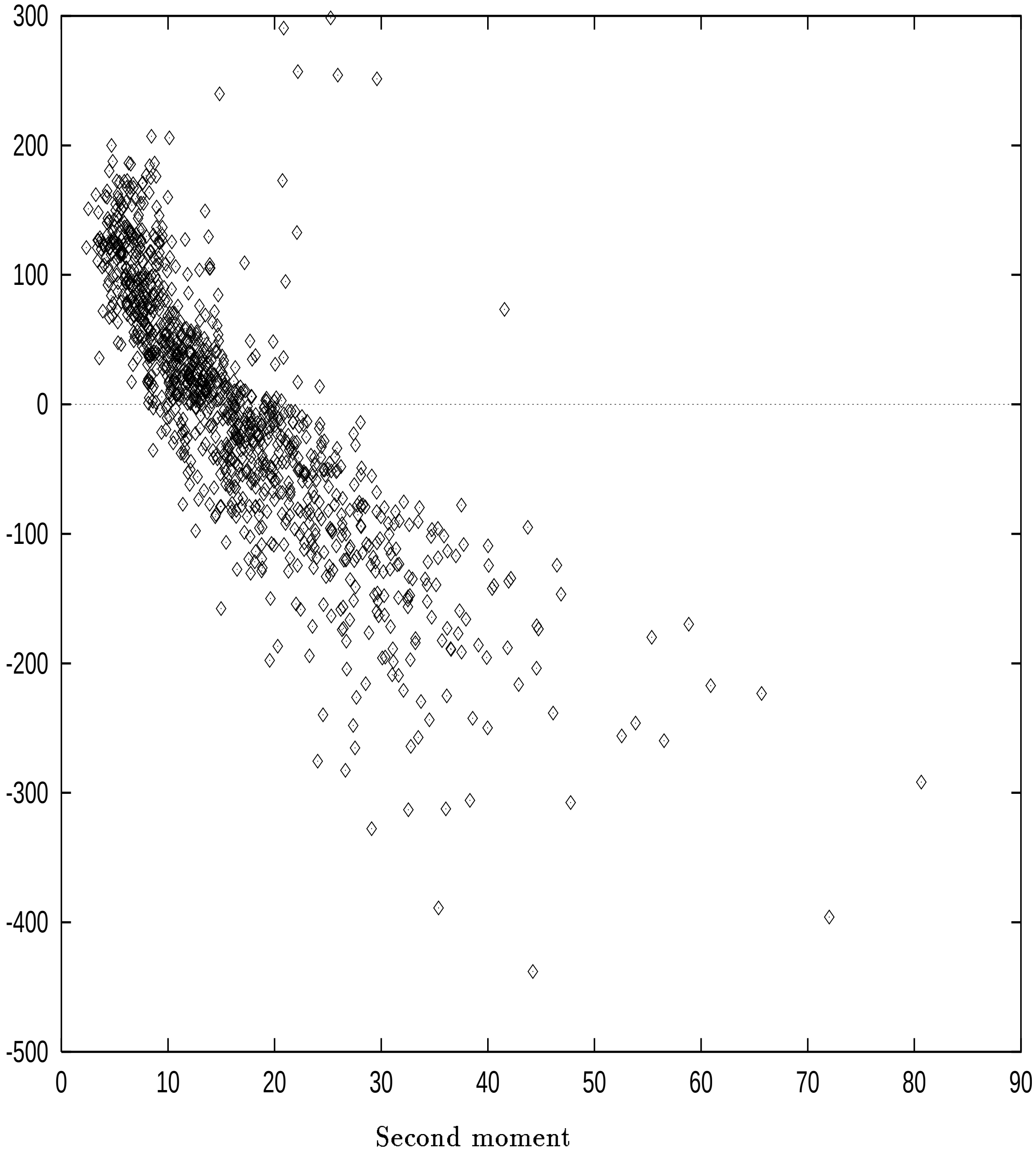

Figure 4.9: Plot of second moment against residuals

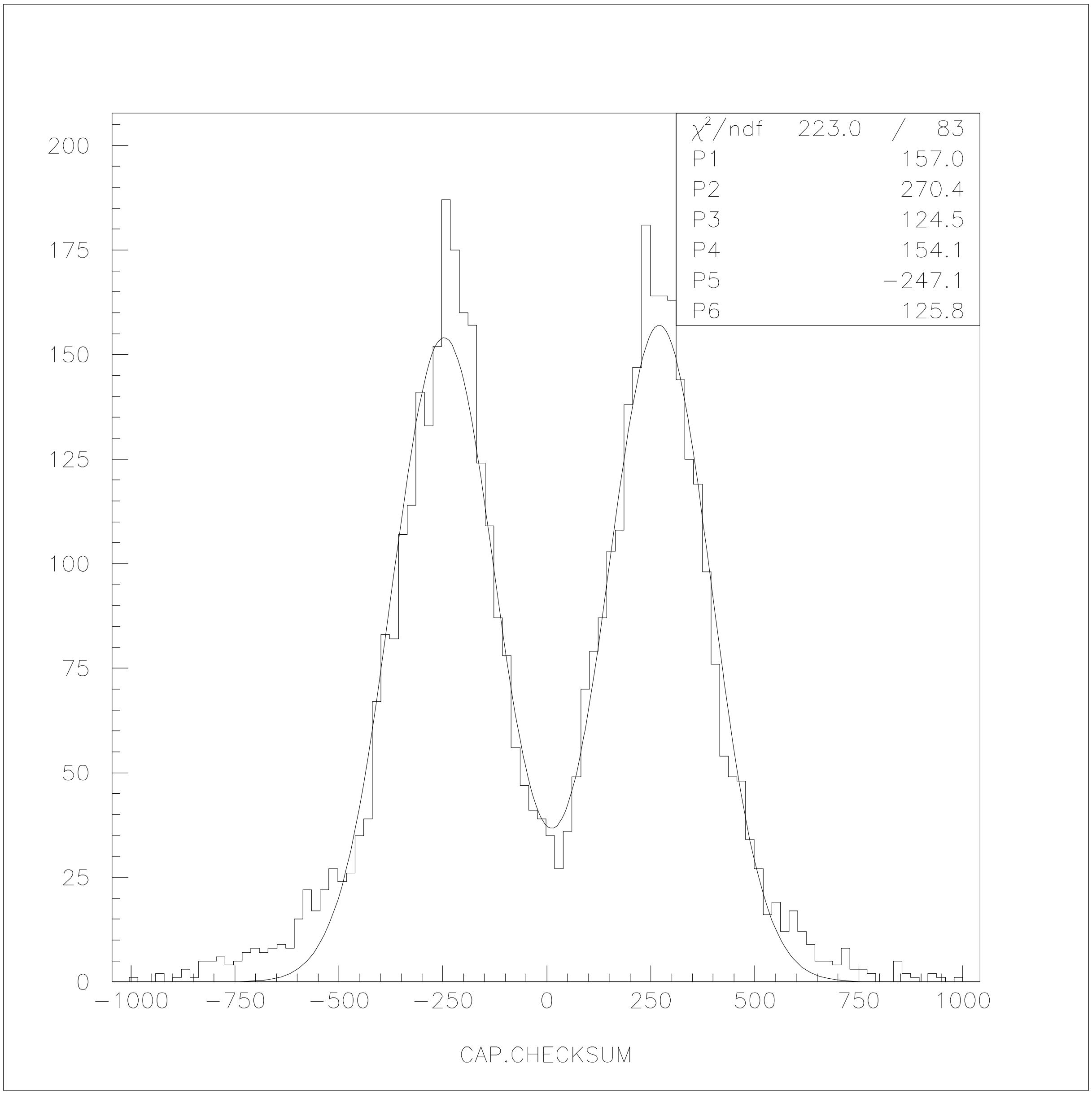

Figure 4.10: Checksum calculated using first and second moments

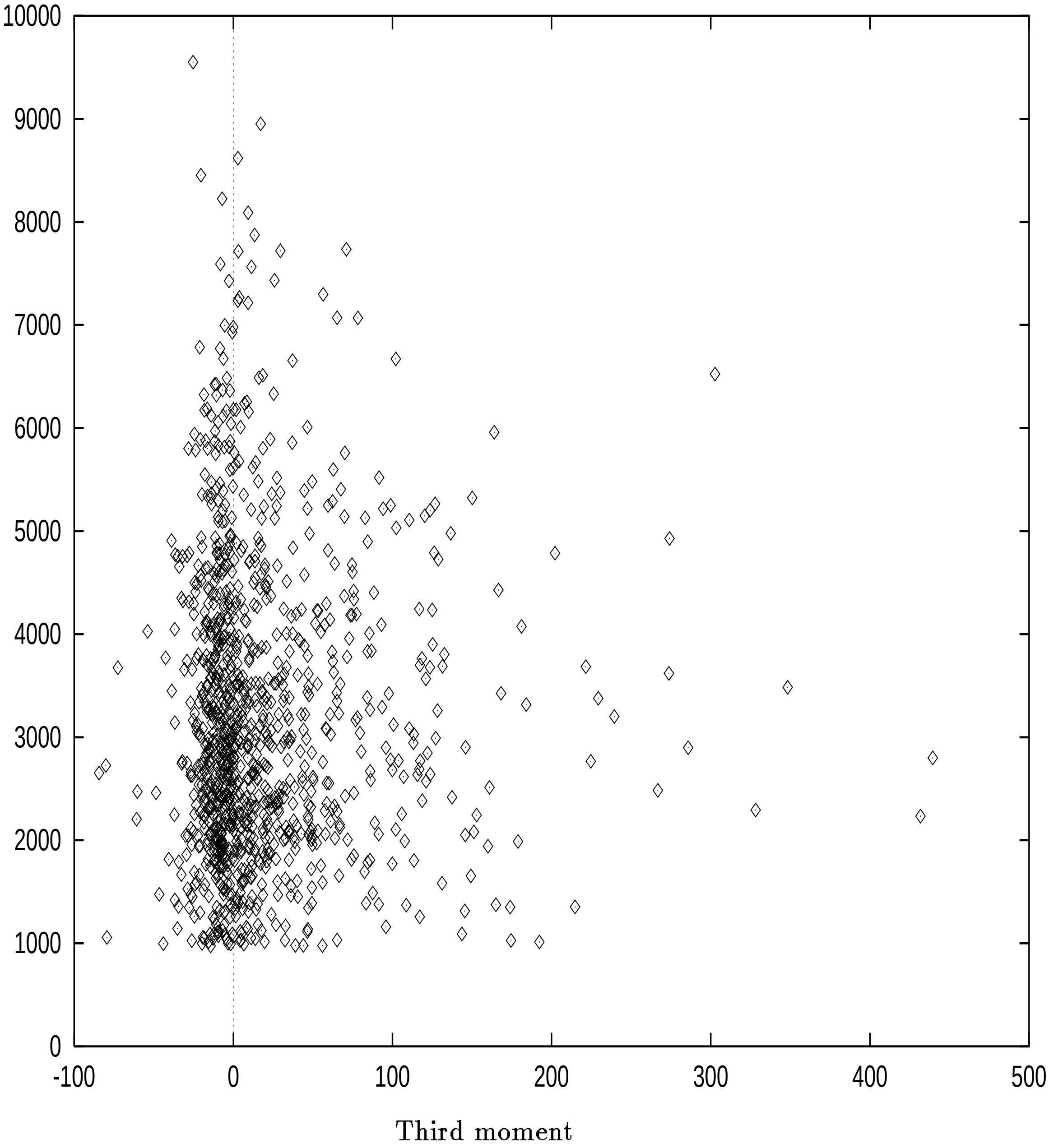

Figure 4.11: Third moment plotted against QT drift time

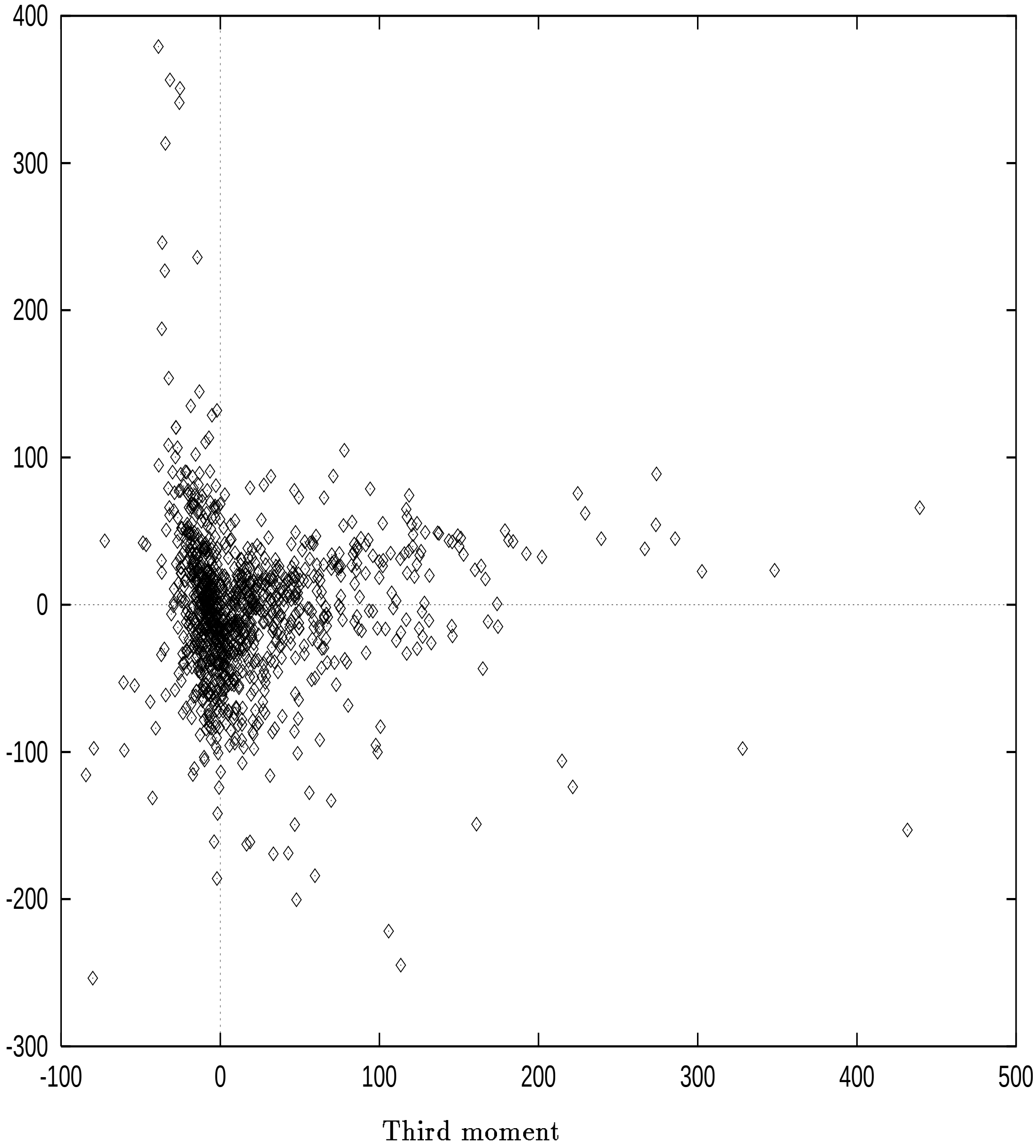

Figure 4.12: Plot of third moment against residuals

Bytes	Resolution
1	124.1, 126.5
2	124.4, 125.8
4	124.5, 125.8

Table 4.3: Width of checksum peak

The amount of bytes needed to store the moments is influenced by two things: the actual resolution required and how the memory is transferred from one detector sub system to another. For example, currently, many systems transfer data in blocks of four bytes, so storing the moment data in less than four bytes on such a system would be wasteful. Table 4.3 shows the resolutions of the checksums when calculated using moments which have been stored in the indicated number of bytes.

When one byte is used to store each moment, the width of the checksum peaks change slightly. Using 1 byte per moment and storing the first two moments would give an average compression gain of approximately 10 or a reduction in data size of 90%. In cases where some loss in resolution can be tolerated, this is a very significant level of compression. Increasing the storage to two bytes per moment and again storing only the first two moments decreases the compression ratio to approximately 5, for a reduction in data size of about 80%. Increasing the accuracy of the stored moments by using more then 2 bytes to store each moment does not effect the checksum resolution.

Chapter 5

Conclusion

5.1 Lossless Compression

The best lossless compression method of all those considered was the Huffman encoding, of the linearised difference of samples. This compressed the digitisations by 35.4%, including the header data required for decompression.

However, using this directly at the output of the digitiser introduces several problems. Firstly, the data transfer methods currently in use transfer a fixed number of bytes per transfer cycle, usually in multiples of 4 or 16 bytes. This would mean that for most compressed digitisations a couple of pad bytes would be required. This padding of the compressed data would reduce the amount of effective compression. Secondly, although Huffman compression is fairly easy to implement, the cost of adding the extra electronics may outweigh the savings gained in reducing the electronics required to connect together the detector sub units.

The Huffman compression scheme is ideally suited to compress data just before it is saved to permanent storage. The compression of 35.4% is clearly better than the 25.1% compression using the best generally available multipurpose compression program, *gzip* . Such compression would be relatively simple to add to current software, by altering the FPACK data store/retrieval routines so that the compression/decompression would be transparent to the user or at least achieved with a single call to a compression/decompression routine.

5.2 Lossy Compression

The method of storing only the positive linearised difference of samples values has some interesting advantages. The compression percentage of approximately 70% indicates a significant saving in size. Further more, it gives the same timing accuracy, since no data needed for the timing calculations is removed. The method does have some drawbacks in that to select the part of the difference of samples used for timing, the QT analysis algorithm uses the linearised sample values. A method would need to be devised so that only positive difference of samples are used to determine which difference of sample values are used to determine the drift time.

Another problem is that different detector sub units use different methods to determine the drift time. Although using the positive difference of sample values could be used for all detector sub units the timing resolution would be poorer than the optimised methods. This method of lossy compression has the advantage of being able to cope with multiple hits within a single digitisation.

In using the moments to calculate the drift time, the resolution on the drift time is poorer. Only the first two moments are significantly correlated with the drift time and give a noticeably poorer drift time resolution. The higher moments are not correlated with the drift time and cannot be used to improve the accuracy of the drift time calculation.

The compression of data using moments is, however, very large. Each moment can be stored in 2 bytes, or 4 bytes for both moments. This gives a compression reduction of approximately 80 percent.

Aside from the poorer resolution, moments have an additional drawback in that the method can not be used directly for digitisations containing multiple hits. Either the method is used only for detector sub units with low a probability for multiple hits or a digitisation would have to be split into sections, each section containing only a single hit. The moments for each section would then be calculated separately.

5.3 Closing Remarks

This study has shown that subject to certain well defined limitations the size of data flow around a detector experiment can be reduced by a factor ranging from 1.5 to 10 depending a whether information is fully retained or partially discarded.

Appendix A

Computer Code

In investigating the various methods for compression of pulse digitisations a large amount of computer code was written. This appendix describes the code and lists the pertinent parts of the code. The program was written in the C programming language, to the ANSI C standard. Data from FPACK files is read in using the FPACK and BOS library routines. Graphical output is performed by piping the program to either *gnuplot* or *paw* depending on the output required. *gnuplot* is a freely available plotting program and *paw* - physics analysis workstation - is a program written by the Computing and Networks Division at CERN.

A macro defined in *header.h* is used throughout the program to ease debugging:

```
#define DEBUG 1 /* 1 for debugging info */

    . . .

#if DEBUG == 1
  #define IFDEBUG(text) text
#else
  #define IFDEBUG(text)
#endif
```

if DEBUG is defined as 1 then the IFDEBUG macro is replaced by the string inside the IFDEBUG brackets otherwise no replacement is made.

The data from the FPACK files is read in on an event by event basis, using the FPACK library routines, in the function *event_read*. The data from a single event is stored in a single structure which contains the raw digitisations, both linearised and unlinearised, and the drift times returned from QT analysis. The structure is defined in *header.h*. After the data for the event has been read in, the function *calc* calculates the moments of each digitisation as well as the position of maximum of the digitisation and the position of the maximum in the difference of samples of the digitisation.

```
/* calculate moments */

for (k = 0; k < 2; ++k)
{
  lv[k] = 255;
  for (j = start; j < revent->dFRRD[irFRRD].size; ++j)
    if (revent->dFRRD[irFRRD].d[k][j] < lv[k]) lv[k] = revent->dFRRD[irFRRD].d[k][j];
  for(j = start; j < revent->dFRRD[irFRRD].size; ++j)
    revent->dFRRD[irFRRD].d[k][j] -= lv[k];
}

for (k = 0; k < 2; ++k)
{
  start = 0;
  end = revent->dFRRD[irFRRD].size;

/* initially set things to zero */
  for (j = 0; j < 4; ++j)
    revent->dFRRD[irFRRD].m[k][j] = 0.0;
  sum[k] = 0;

/* calculate the first moment */
  for (j = start; j < end; ++j)
  {
    l = j + revent->dFRRD[irFRRD].istart;
    sum[k] += revent->dFRRD[irFRRD].d[k][j];
    revent->dFRRD[irFRRD].m[k][0] += revent->dFRRD[irFRRD].d[k][j] * l;
  }
  revent->dFRRD[irFRRD].m[k][0] = revent->dFRRD[irFRRD].m[k][0]/sum[k];

/* calculates moments 2 to 4 */
  for (j = start; j < end; ++j)
  {
    l = j + revent->dFRRD[irFRRD].istart;
    v = l - revent->dFRRD[irFRRD].m[k][0];
    revent->dFRRD[irFRRD].m[k][1] += (v * v)*revent->dFRRD[irFRRD].d[k][j];
    revent->dFRRD[irFRRD].m[k][2] += (v * v * v)*revent->dFRRD[irFRRD].d[k][j];
    revent->dFRRD[irFRRD].m[k][3] += (v * v * v * v)*revent->dFRRD[irFRRD].d[k][j];
  }
  revent->dFRRD[irFRRD].m[k][1] /= sum[k];
  revent->dFRRD[irFRRD].m[k][2] /= sum[k];
  revent->dFRRD[irFRRD].m[k][3] /= sum[k];
  revent->dFRRD[irFRRD].sum[k] = sum[k];
}
```

```
for (k = 0; k < 2; ++k)
{
  for(j = start; j < revent->dFRRD[irFRRD].size; ++j)
  revent->dFRRD[irFRRD].d[k][j] += lv[k];
}
```

First the value the of the smallest time slice is subtracted from the value of each of the time slices in the digitisation. The variables *start* and *end* allow for the calculation of the moments to be performed over only part of the digitisation, but the best results used the whole of the digitisation to calculate the moments.

A simple, but memory intensive, set of functions was written to implement Huffman encoding. First the *huffman_prob* function counts the number of occurrences of each value then *huffman_tree* is called to create the Huffman tree.

```
void addstr(int p[COMPMAX], int l[COMPMAX][COMPMAX], char c[COMPMAX][COMPMAX],
            int v, char *a)
/*
    add the character a to the string c[v] and any other code string listed
    in l[v][..]
*/
{
  int i, j;

  strcat(c[v], a);
  for (i = 0; l[v][i] != -1; ++i)
  {
    j = l[v][i];
    addstr(p, l, c, j, a);
    }
}

    . . .

void huffman_tree(int p[COMPMAX], int l[COMPMAX][COMPMAX], char c[COMPMAX][COMPMAX],
                  int x, int y)
/*
    construct Huffman tree
*/
{
  int i;
  int fin;
  int low1v, low1p;
  int low2v, low2p;

  fin = 0;
  do
  {
/* find the lowest probability */
    low1v = -1;
    low1p = -1;
    for (i = x; i <= y; ++i)
```

```
    {
      if (p[i] != -1)
      {
        if (low1v == -1 || p[i] < low1v)
        {
          low1v = p[i];
          low1p = i;
        }
      }
    }

/* find the second lowest probability */
    low2v = -1;
    low2p = -1;
    for (i = x; i <= y; ++i)
    {
      if (p[i] != -1 && i != low1p)
      {
        if (low2v == -1 || p[i] < low2v)
        {
          low2v = p[i];
          low2p = i;
        }
      }
    }

    if (low1p != -1 && low2p != -1)
    {
      addstr(p, l, c, low1p, "0");
      addstr(p, l, c, low2p, "1");
      for (i = 0; l[low2p][i] != -1; ++i)
        ;
      l[low2p][i] = low1p;
      p[low2p] += p[low1p];
      p[low1p] = -1;
    }
    else fin = 1;
  } while (fin == 0);
}
```

The function is called with *p[COMPMAX]* containing a count of the number of occurrences of each value, x and y are the limits on the valid range of values. *l[COMPMAX][COMPMAX]* is an array used to construct the code and the code is returned in the array *c[COMPMAX][COMPMAX]*. The data is then stored in memory using the following code:

```
void huffman_store(int b[COMPMAX][2], int *base, int *comp, int opt, int start)
{
  int i, j, k, l;
  int e, bytes;

  *base = 0;
  *comp = 0;
  e = 0;
  if (opt == 2) e = -1;
```

```
  while (read_event(&event, save_flag) == 0)
  {
/* if start equals 1 store compressed data in a single block per event */
    if (start == 1) start_push((void *) cblock);
    for (i = 0; i < event.noFRRD; ++i)
    {
/* if start equals 2 store compressed data in a separate block from each wire */
      if (start == 2) start_push((void *) cblock);
      if (start == 1 || start == 2) compress_push_header(&event, i);
      for (k = 0; k < 2; ++k)
      {
/* if start equals 3 store the compressed data for each digitisation separately */
        if (start == 3)
        {
          start_push((void *) cblock);
          push_bits(event.dFRRD[i].size, 8);
        }
        if (opt == 2) push_bits(event.dFRRD[i].u[k][0], 8);
        for (j = 0; j < event.dFRRD[i].size + e; ++j)
        {
          switch (opt)
          {
/* store unlinearised digitisations */
            case 1:
              l = event.dFRRD[i].u[k][j];
              push_bits(bitc[l][0], bitc[l][1]);
              break;
/* store unlinearised DOS */
            case 2:
              l = event.dFRRD[i].u_dos[k][j] + 255;
              push_bits(bitc[l][0], bitc[l][1]);
              break;
            default:
              terminal_error("unknown option to huffman_store");
              break;
          }
        }
        if (start == 3) *comp += end_push();
      }
      *base += event.dFRRD[i].size*2;
      if (start == 2) *comp += end_push();
    }
    if (start == 1)
    {
      bytes = end_push();
      *comp += bytes;
      save_event(&event, cblock, bytes);
    }
  }
}
```

The functions *start_push*, *push_bits* and *end_push* handle the writing of variable numbers of bits at a time to a continuous memory block as shown below.

```
/* compression library routines */

/* comlib.c */
```

```
#include "header.h"
#define MAX_SIZE 1024

/*
    routines to store/retrieve data from an area of memory, where the
    data is a stored as a set of bits of arbitrary length

    Limits: maximum bits stored/retrieved in one operation: no of bits
    in an integer minus 7, for hpux = 25 bits
*/

static unsigned char *push_pos;
static unsigned int push_word;
static int push_bc;
static int push_bs;
static unsigned char *pull_pos;
static unsigned int pull_word;
static int pull_bc;
static int pull_bs;

void start_push(void *pos)
/*
    start the push operation at the position pos
*/
{
  push_pos = pos;
  push_word = 0;
  push_bc = 0;
  push_bs = 0;
}

int end_push(void)
/*
    end the push operation, pushing an incomplete byte into the memory
    area if necessary and return the number of bytes pushed
*/
{
  int i;

  i = push_bs;
  if (push_bc > 0)
  {
    push_c(push_word & 255);
    ++i;
  }
  push_pos = NULL;
  push_word = 0;
  push_bc = 0;
  push_bs = 0;

  return(i);
}

void push_bits(int data, int bits)
/*
    push the first bits of the integer data into the memory area
*/
{
  data = data & ((1 << bits)-1);
  push_word += (data << push_bc);
  push_bc += bits;
  while (push_bc > 7)
```

```
  {
    push_c(push_word & 255);
    push_bc -= 8;
    push_word = (push_word >> 8);
  }
}

void push_data(int data, int bits)
/*
    push the first bits of the a signed integer data into the memory area
    and push a sign bits into the memory area in addition
*/
{
  if (data < 0) push_bits(1, 1), data = - data;
  else push_bits(0, 1);

  push_bits(data, bits);
}
```

References

[1] ALTAS Collaboration, *Technical Proposal for the ATLAS detector*, CERN, 1994

[2] H1 Collaboration, *Technical Proposal for the H1 detector*, DESY, March 1985

[3] H1 Collaboration, *The H1 detector at HERA*, DESY 93-103, July 1993

[4] D Mercer et al., *The H1 Scanner M1070*, University of Manchester, UK, July 1990

[5] S Kolya, *H1 Drift Chamber Data System*, DESY, October 1992

[6] S. Burke et al., *Track Finding and Fitting in the H1 Forward Track Detector*, H1-03/95-434, March 1995

[7] V. Blobel, *The BOS System*, DESY Internal Report R1-88-01, January 1988

[8] P. Binko et al., *F-PACK*, Internal H1 Software, August 1994

[9] Burkhard Burow, *cfortran*, CERN PROGRAM LIBRARY, Release 204, 1993, Available by anonymous ftp from zebra.desy.de

[10] D. A. Huffman, *A method for the construction of minimum redundancy codes*, Proceedings of the IRE, 40 1098-1101, 1952

[11] C. E. Shannon, *A Mathematical Theory of Communication*, Bell Syst. Tech. J., 27 398-403, July 1948

[12] G. G. Langdon, *An Introduction to Arithmetic Coding*, IBM J. Res. Develop., 135-149, March 1984

[13] J. J. Rissanen, *Generalised Kraft Inequality and Arithmetic Coding*, IBM J. Res. Develop., 20 198-203, 1976

[14] A. C. Popet, *Scalar Quantisation With Arithmetic Coding*, MSc. Thesis, Massachusetts Institute of Technology, June 1990

Acknowledgements

I would like to thank all the people who have greatly helped with the production of this thesis.

Particular thanks go to my supervisor, Robin Marshall, for his constant help throughout my time at Manchester, and for his many constructive comments on the writing of this thesis, which he patiently ploughed through on several occasions.

I would also like to thank Joe Foster for putting up with my many questions about H1 software and for his many good ideas about this project.

Finally many thanks must go to the other members of the Manchester HEP group, fellow students and staff, with whom working was a pleasure.

TEST LINE

QT drift time

QT drift time

Residuals

QT drift time

Residuals