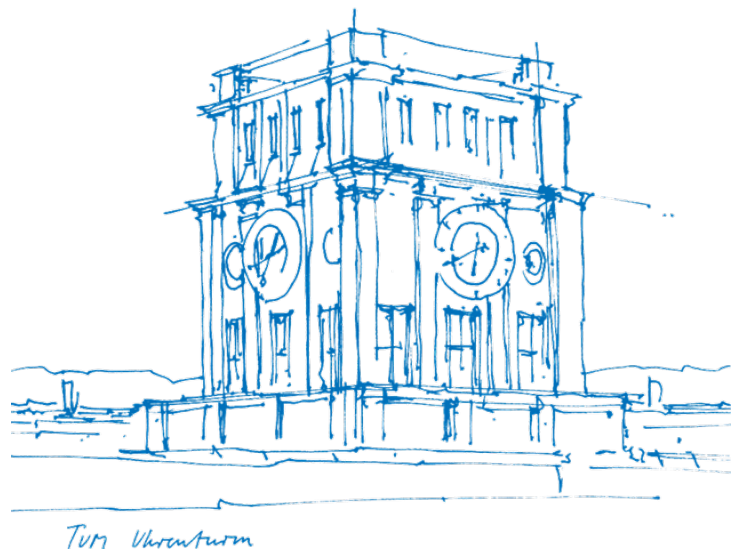


Event Camera Lossless Compression for Satellite Applications

Event Data Compression on a Nvidia Jetson Orin Nano for CubeSats

Antonio Junco de Haas



Event Camera Lossless Compression for Satellite Applications

Event Data Compression on a Nvidia Jetson Orin Nano for CubeSats

Antonio Junco de Haas

Thesis for the attainment of the academic degree

Master of Science (M.Sc.)

at the TUM School of Engineering and Design of the Technical University of Munich.

Examiner:

Aliakbargolkar, Alessandro

Supervisor:

Dolan, Sydney

Submitted:

Munich, 14.10.2025

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Munich, 14.10.2025

Antonio Junco de Haas

Abstract

This thesis will dive into the details of the event camera lossless compression algorithm for the EventSat mission. EventSat is a Low Earth Orbit CubeSat mission that will utilize an event camera to observe stars and satellites. These cameras only record changes in pixel brightness. At the time of writing, there are no event data compression algorithms that are optimized for space applications. The algorithm's objective is to minimize file size and maintain data integrity for downlink purposes. The algorithm compresses data with multiple strategies, namely delta, dictionary, and variable-length encoding. Histogram analysis automatically calculates optimal algorithm thresholds. Additional compression strategies, such as bit splicing and pair encoding, are also implemented. The compression ratio is up to 84%, surpassing generic compression algorithms. The algorithm is open-sourced to support further research.

Contents

Abstract	vii
1 Introduction	1
1.1 Research Contributions	1
2 Literature Review	3
2.1 Mission Overview	3
2.2 Space Situational Awareness	3
2.3 Event Cameras	5
2.4 Lossless Compression	5
2.4.1 File Integrity	5
2.5 Sorting Algorithms	6
2.6 Delta Encoding	7
2.7 Dictionary Based Encoding	7
2.8 Pair Encoding	7
3 Methodology	9
3.1 Event Camera Data	9
3.1.1 Event Data Entries Examples	9
3.1.2 Event Queues	11
3.2 Techniques and Strategies	11
3.2.1 Index Header Data Structure	11
3.2.2 Index Header Compression	11
3.2.3 Sorting Algorithm	12
3.2.4 Delta Encoding	13
3.2.5 Pair Encoding	16
3.2.6 Dictionary Encoding	19
3.2.7 Variable Length Encoding	21
3.2.8 Unsigned Integers	23
3.2.9 Custom Integer Length	23
3.2.10 Bit Splicing	24
3.3 Techniques and Strategies: Summary	24
3.4 Algorithm Structure	26
3.4.1 Compression	26
3.4.2 Decompression	27
4 Results	29
4.1 Compression Sizes and Datasets	29
4.2 Changing Index Compression Settings	31
4.3 Ablation Studies	32
5 Conclusions	33
5.1 Limitations	33
5.2 Future Work	33
5.3 Final Conclusions	33

1 Introduction

Satellites can be considered as a type of computer in space that can communicate back to Earth. The physical location of space offers a variety of advantages to satellites; for example, an uninterrupted line of sight allows telecommunications to Earth, while the lack of atmosphere allows for clear space observations. Satellites can vary in size based on their mission objectives. One of the most popular new satellite types for universities is the CubeSats (Cube Satellites). These specific satellites are made of lightweight Units (U) that are $10\text{cm} \times 10\text{cm} \times 10\text{cm}$. By standardizing sizes, the satellite development is simplified. Additionally, some ride-sharing companies utilize CubeSats as standard, making rocket launches affordable by reducing costs.

EventSat is a university satellite mission that will observe stars and other space objects. The satellite is a 6-unit (6U) Low Earth Orbit (LEO) CubeSat mission developed by the TUM Chair of Spacecraft Systems [8]. The general objective of the mission is to observe stars, planets, interplanetary objects, other satellites, and space debris. The satellite will observe objects in real-time and transmit recordings to a database on the ground.

EventSat will utilize an event camera to collect observations. These sensors function similarly to human eyes by recording brightness changes of individual pixels, rather than the whole frame. The camera can observe fast-moving objects, which may appear blurry or dim in traditional picture cameras. Since the satellite is already in space, the image quality will improve, as the satellite will have little to no atmospheric image interference.

While in-space observations are beneficial to the payload, space also introduces challenges for the mission. In space, there are limited communication windows with satellite operators on the ground, as the satellite moves quickly across the sky. The downlink windows are too short to transmit all recorded observations in their entirety from the satellite. As a result, information sent to and from the satellite must be clear, concise, and compressed. By compressing the original files, the satellite will be able to send more information in a shorter amount of time.

This thesis will focus on lossless data compression algorithms associated with the EventSat mission. Companies that create event cameras also offer a single solution software for data compression; as such, additional interest in compression algorithm development is limited. At the time of writing, there exist very few compression algorithms focused on the event data format. The file sizes produced by existing general-purpose compression techniques are fairly large. For a space-focused application, it would be ideal if the size were reduced further, as event cameras can generate millions of events in under a minute, which would take a significant amount of time to downlink. Furthermore, directly applying existing general-purpose compression algorithms is complicated because it may require fine-tuning the algorithm settings for the values recorded and observation length. If implemented as described above, development and execution would be time-consuming, and the mission would be reliant on third-party software. Additionally, these algorithms are not optimized for space applications, as they may prioritize execution speed rather than compression ratio or low-resource utilization.

1.1 Research Contributions

The purpose of this thesis was to create a compression algorithm optimized for event camera data in space applications. The contributions are:

- A novel event data compression algorithm that uses a hybrid dictionary and variable-length encoding strategies. The dictionary encoding approach categorizes the data by size and compresses it

in a separate section from the rest of the file. The variable-length encoding approach integrates compression instructions, indicating size right next to the data.

- A comprehensive trade-off histogram analysis to determine the optimal compression thresholds for dictionary and variable length. The algorithm calculates file sizes for each threshold until the minimum size is found.
- Bit-splicing to reduce the amount of data padding present in standard-length integers. This strategy allows the algorithm to write information in files at the sub-byte level.
- Low-resource algorithm architecture to limit memory and the number of operations on board and on the ground. The algorithm development considers that resources are limited and prioritizes efficient operations.

All data and code associated with this project are open source to facilitate further study.

The thesis is organized as follows. In Chapter 2, we provide a brief mission background and literature review of the algorithms used in this thesis. The main algorithm will focus on lossless compression. Chapter 3 explains the implementation of the algorithm. A table is included to demonstrate the trade-offs and interactions between strategies. Next, Chapter 4 quantifies the algorithm execution. Compression strategies are evaluated based on their compression ratios and the ability to maintain data integrity. Ablation Studies show how each strategy slowly compresses file sizes until a minimum is reached. Finally, Chapter 5 summarizes the contributions of this thesis, limitations of this work, and identifies key areas of future work.

2 Literature Review

The literature review presents the background research relevant to the thesis. Section 2.1 summarizes the mission overview for EventSat, briefly mentioning the mission objectives as well as highlighting the need for the compression algorithm. Next, Section 2.2 introduces Space Situational Awareness (SSA), a relevant field of study within the space sector. Furthermore, 2.3 introduces what an event camera is and its relevance for SSA. Additionally, Section 2.4 introduces the concept of lossless compression and its relation to file integrity.

Afterwards, the section will delve into algorithms that will assist with the compression. The algorithms are sequentially ordered as they will be utilized in the final compression executable. Firstly, Section 2.5 discusses the sorting algorithms that will help with moving events with similar values. Next, Section 2.6 provides a brief description of delta encoding, which encodes only the differences of values. Then, Section 2.7 introduces the dictionary encoding and the histogram analysis for optimal event lengths. Lastly, pair encoding is explained in Section 2.8, where two numbers are compressed into a single one.

2.1 Mission Overview

EventSat consists of two Mission Objectives: to develop a database of space-based event camera observations and perform real-time detection to identify objects in the event camera's field of view using computer vision. The Concept of Operations for Mission Objective 1 and Mission Objective 2 can be viewed in Figures 2.1 and 2.2. The relevant modes for this thesis are the Observation Mode, where data is generated, and the Compression Mode, where the data will be reduced in size.

To properly accomplish the mission objectives, a high amount of data must be downlinked and stored by the ground station. Current projected estimates place the required database size at 180 MB [7]. Downlink capabilities will be limited due to the communication window available in the satellite's orbit. Information may also need to be resent, as packets of data could be lost during transmission.

To ease the data transmission restrictions, compression algorithms will be developed from scratch. Making the algorithms from the beginning allows them to be optimized for space applications. For example, decompression has no impact on the satellite, as it is executed on the ground. Additionally, because communication windows are limited, the compression doesn't need to be real-time. If traditional compression algorithms for event data were to be utilized, it may result in larger files. When those files are set to downlinked, more time will be needed, and thus limit further observations.

Conducting event camera observations is the duty of the payload subsystem. Data will be managed by an Nvidia Jetson, where post-processing algorithms for lossy and lossless compression will be executed. The lossless compression algorithm for this thesis will be run here. Afterwards, additional procedures are run and the file is sent to the antenna, where it will be downlinked to the mission's ground station.

To downlink data, certain requirements must be met. First, data must be ready to be downlinked; all post-processing procedures must be finished by then. Secondly, the satellite is passing over the ground station. These communication windows last a few minutes and occur multiple times per day. Data packets sent to the ground station may be lost or corrupted; therefore, the downlink capabilities may be reduced. Additionally, file integrity verifications must be performed to prove that the file was sent correctly.

2.2 Space Situational Awareness

EventSat will observe, detect, and categorize objects in space. These actions are part of the Space Situational Awareness (SSA) study area. SSA is broadly defined as the knowledge of surrounding space

Mission Objective 1: Develop a database of space-based event camera observations

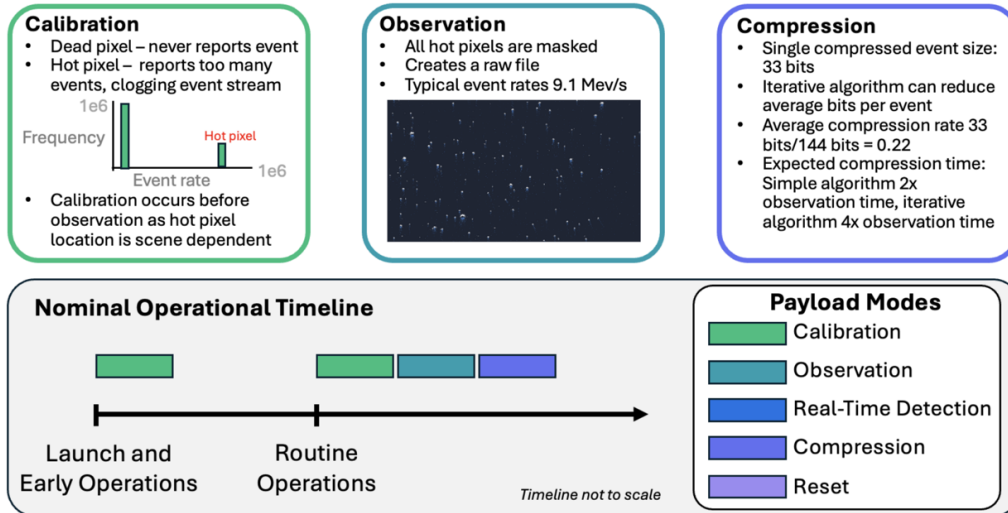


Figure 2.1 Mission Objective One Concept of Operations.

Mission Objective 2: Perform real-time detection to identify objects in the event cameras field of view using computer vision

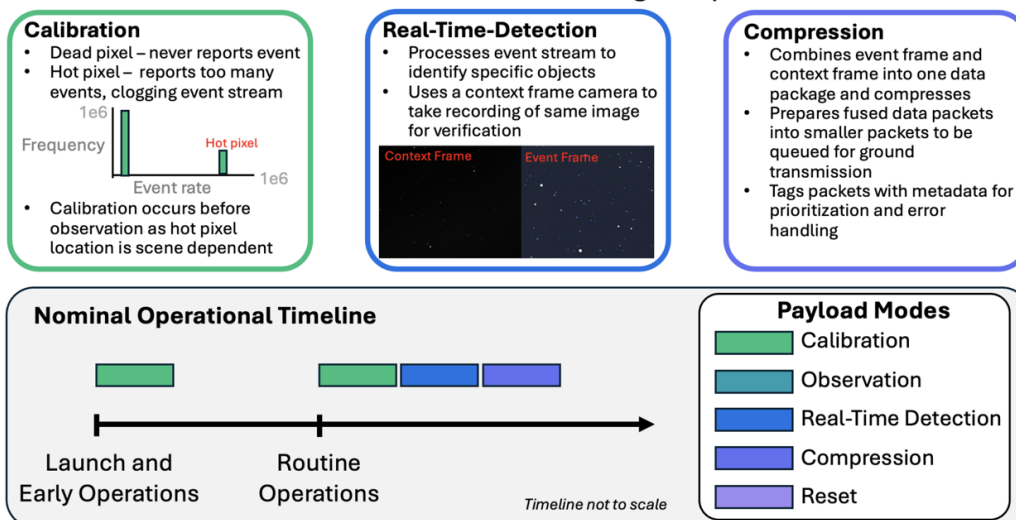


Figure 2.2 Mission Objective Two Concept of Operations.

phenomena, natural and human-made [14]. The European Space Agency has an SSA Programme with the purpose of understanding the space environment, especially space weather, near-Earth objects, space debris, and forecasting related events. [11]. Understanding the near-space environment is crucial for protecting satellites from natural and human-based disasters.

2.3 Event Cameras

Existing research has suggested that event cameras (ECs) have the potential to surpass traditional frame-based cameras [22]. The key difference between ECs and other types of sensors is that ECs are neuromorphic sensors that register intensity changes as 'events' with ultra-low latency. These events are composed of four measurements: x and y coordinates of a pixel, the polarity p (brightness increase or decrease), and the timestamp t . Events can be quickly recorded, allowing the sensor to capture fast-moving objects that may go undetected by a traditional camera.

ECs are being considered for SSA [25], because their characteristics are beneficial for space applications. The first is High temporal Resolution, events are detected and recorded within milliseconds. In the space environment, measurements will have millisecond precision. The next advantage is Low Latency, because pixels operate independently from each other, with shorter reset periods between firing. Specifically for SSA applications, an event camera can detect incoming information in real time, without waiting for the camera to complete a frame. Furthermore, ECs can operate with High Dynamic Range, exceeding $60dB$. This is an advantage in the space environment, as brightness changes can vary in range; very bright objects, as the Sun, can oversaturate images in traditional cameras. Additionally, Low bandwidth, as only the delta brightness is recorded, reduces data transmission requirements for satellites. Finally, Low-power cameras can operate in the mW range, ideal for satellites with power constraints. These preliminary results contribute to the motivation for using an Event Camera for EventSat.

2.4 Lossless Compression

An investigation into current Lossless Compression algorithms for event data was conducted. However, the only relevant results were proprietary compression algorithms developed by the event camera manufacturers, such as the .raw files generated by Prophesee [21]. Due to the limitations of the existing algorithms, EventSat considers it relevant to utilize its own lossless compression algorithm.

Compression techniques in space differentiate between lossy and lossless compression. Lossy compression removes unwanted data from the file. By eliminating information, the file becomes smaller. Lossless compression reduces file size without deleting any data. After decompression, the file remains intact. This thesis focuses solely on lossless compression.

Lossless compression algorithms maintain data integrity. Values can be modified several times, but the file should be able to be reconstructed without any changes to the original data. At this point, all the data in the file is considered significant and thus should be received by the ground station.

Compression algorithms have certain parameters that can be modified to achieve better compression ratios or faster execution speed [3]. For EventSat, the priority is a higher compression ratio for data downlink. However, onboard resources are limited, so the algorithm must be efficient and have a low number of operations.

2.4.1 File Integrity

File integrity refers to maintaining the original file intact [9]. Integrity checks have a wide variety of applications in networks and compression. Validating the integrity of a file will guarantee that the compression algorithm executed correctly. Integrity checks are relevant for data downlink, as they verify that the file was correctly reconstructed in the ground station.

A popular file integrity strategy used for file transmissions is creating a fixed-length hash key [23]. The file is run through the hashing algorithm and creates a unique key based on its contents. After the file is compressed and decompressed, the hashing algorithm is run once more. If both hashes are identical, file integrity is maintained.

2.5 Sorting Algorithms

One of the key tools that will allow the algorithm to manipulate the data is the sorting algorithm. These algorithms receive as input numbers that are in disorder and order them sequentially. To measure performance, the worst-case scenario is utilized [16]. Careful consideration when selecting an algorithm is beneficial for memory and time constraints.

To analyze the efficiency of a sorting algorithm, the standard utilized is Big \mathcal{O} notation for the worst-case sorting scenario. Algorithm performance is measured as if the number of elements extends to infinity. The worst-case scenario is when the maximum number of operations is needed to sort the data.

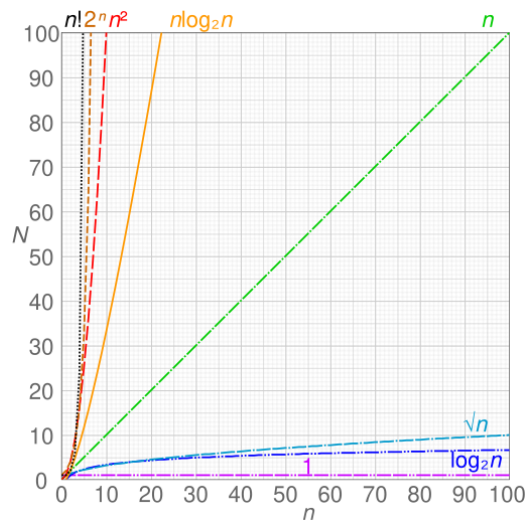


Figure 2.3 [6] N means the number of operations needed to sort, n is the number of input elements.

The two main sorting algorithm types are the slow comparative and the fast non-comparative. In comparative sorting, the algorithm's values must be matched to all other values present in the data. Non-comparative sorting can skip this step and sort the values without comparing the data against the rest. Because of this fundamental distinction, non-comparison algorithms outperform comparison algorithms [10].

However, non-comparison algorithms can only be utilized under certain contexts. For example, counting sort executes in linear time if the data entries exist within a finite set. All elements are integers from the range 1 to k , where k is a finite integer. The elements are then inserted directly into a location in memory respective to their individual value of x . In this procedure, the elements are not compared with each other; rather, they are assigned to a space in memory corresponding to their value [2]. If the event data is structured in a certain way, it may be possible to utilize a non-comparison algorithm and reduce the number of operations within the satellite.

Some popular sorting algorithms and their worst-case scenarios are presented in Table 2.1. Bubble and Insertion Sort are among the most straightforward sorting algorithms. They are easy to implement and perform well when there are a few entries, but become unusable for large datasets. Quick-Sort and Merge-Sort are more complex in structure, but operate temporally faster. However, the worst-case for Merge sort is still $\mathcal{O}(n^2)$; as such, Merge sort outperforms it in this metric. Finally, Counting and Bucket Sort are non-comparison, and as such, they can be solved in linear time. Outperforming the rest of the sorting algorithms in this table.

Sorting Algorithm	Worst-case Big \mathcal{O} Notation	Description
Bubble Sort [26]	$\mathcal{O}(n^2)$	Comparison
Insert Sort [12]	$\mathcal{O}(n^2)$	Comparison
Quick Sort [27]	$\mathcal{O}(n^2)$	Comparison
Merge Sort [1] [15]	$\mathcal{O}(n \log(n))$	Comparison
Counting Sort [20]	$\mathcal{O}(n + k)$	Non-Comparison
Bucket Sort [1]	$\mathcal{O}(n + k)$	Non-comparison

Table 2.1 Comparison and non-comparison sorting algorithm based on big \mathcal{O} complexity.

2.6 Delta Encoding

Delta encoding calculates the difference (delta) between similar data entries [24]. Many algorithms differ in execution speed and compression rate. Due to the nature of the mission, the desired algorithm should have the highest compression ratio without regard to execution time.

The algorithm only stores the difference in the recorded values. For example, encoding the following sequence of numbers [0, 5, 11, 18, 27, 37, 42] would result in the following array [0, 5, 6, 7, 9, 10, 5]. These numbers are smaller and thus require fewer bits to be represented. This is especially true when numbers have many digits, but only have small delta increments in between measurements. Table 3.9 shows how timestamps can be significantly compressed. Measurement increments are typically 1-3 digits, while the overall number is above 7 digits.

The main downside of Delta Encoding is that the data entries become dependent on their position. If one entry is moved, the rest of the file will have a displacement error. A recurring trade-off for compression is the loss of data flexibility and redundancy. The more compressed a file is, the harder it will be to manipulate the data.

2.7 Dictionary Based Encoding

Event Data Entries may have different lengths. Dictionary-based compression [4] [18] rearranges data structures in a different section within the file [18]. By rearranging and keeping track of original locations, data integrity can be maintained [5].

Dictionary-based encoding requires several conditions to perform effectively. First, the algorithm performs better when the data entries have different lengths. Secondly, an identification number must be introduced to mark the original place where the data was found. The data can then be moved to a designated dictionary encoded zone at the beginning or end of the file. This separates long data entries (Dictionary Encoded) from the rest of the entries, which are of a smaller length.

To select which data should be dictionary-encoded, an arithmetic histogram analysis must be executed. The purpose is to find the optimal length for the events. It is a trade-off analysis between the size of the dictionary code block and the leftover information.

2.8 Pair Encoding

Pair encoding is a mathematical concept where two Natural numbers can be combined and represented as a single one. A famous procedure is the Cantor Pairing Function $\pi : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$. Two numbers are encoded as a single one [19].

$$z = \pi(x, y) = \frac{1}{2}(x + y)(x + y + 1) + y \quad (2.1)$$

where $x, y \in 0, 1, 2, \dots$

To invert the Cantor pairing function, we use an intermediate value

$$r = \frac{\sqrt{8z+1} - 1}{2} \quad (2.2)$$

To depair the cantor pairing the following equations are used [13]:

$$x = \frac{r(r+3)}{2} - z \quad (2.3)$$

$$y = z - \frac{r(r+1)}{2} \quad (2.4)$$

It follows that if a bijection can uniquely encode two natural numbers into a single one, the process can then be repeated for a third number. As such, using the event data entries mentioned above 3.1.1, it would be possible to represent the x coordinate, the y coordinate, and the polarity of an event as a single unique number.

3 Methodology

This Chapter explains what the algorithm will read, execute, and output. Section 3.1 presents some example entries of raw event data and how the events are stored within the algorithm, namely the index and timestamp. Next, Section 3.2 will go into detail regarding the strategies the algorithm will utilize, as well as the required analysis. Additionally, Section 3.3 summarizes the interactions between encoding strategies. Lastly 3.4 illustrates the step-by-step procedure of compression and decompression.

3.1 Event Camera Data

This Section introduces examples of event data in Subsection 3.1.1, and explains what each number represents. Additionally, it introduces data structures of manipulated events that will assist with the file compression. Next, 3.1.2 briefly introduces efficient methods for storing event information.

3.1.1 Event Data Entries Examples

To properly develop the compression algorithm, some test data was utilized. The EventSat payload team provided a few data sets to help achieve this goal. The main file utilized is "60_DegreePanPerMinute (Berlin).txt". The file was generated by simulating the night sky of Berlin with a 60-degree pan per minute on Stellarium. The data were then obtained by filming a television screen with the event camera. This also considers the camera's field of view. Table 3.1 shows the first events recorded in the file.

%geometry:1280,720
189,513,0,13958339
640,467,1,13958368
887,102,0,13958570
243,693,0,13958593
674,220,0,13958653
1075,510,0,13958664
494,676,1,13958757
304,606,0,13958795
699,411,0,13959103
...

Table 3.1 Raw data from "60_DegreePanPerMinute (Berlin).txt"

While initially it may not be clear exactly what the numbers mean, the header can clarify what each entry represents (Table 3.2).

The data will now be categorized as Event id id , index i , and timestamp t . The id is the number of the event as in the original file, without any sorting algorithm employed. The Index is obtained by combining the x coordinate, y coordinate, and p polarity. Lastly, the Timestamp, which shows when the event occurred. The new structure can be observed in Table 3.3 below.

The Event id is a data structure used for compression; it helps illustrate how the compression algorithm sorts and resorts events so procedures are easier to understand. The main issue with including an Event id is that as data grows, the id will become longer; it may not be optimal to include it in the file. After lossy compression, the maximum number of events in the test data was around 6.7 million for the recording.

x coordinate	y coordinate	polarity	timestamp
189	513	0	13958339
640	467	1	13958368
887	102	0	13958570
243	693	0	13958593
674	220	0	13958653
1075	510	0	13958664
494	676	1	13958757
304	606	0	13958795
699	411	0	13959103
...

Table 3.2 Event data separated into number categories.

<i>id</i>	Index <i>i</i>			<i>t</i>
	<i>x</i>	<i>y</i>	<i>p</i>	
1	189	513	0	13958339
2	640	467	1	13958368
3	887	102	0	13958570
4	243	693	0	13958593
5	674	220	0	13958653
6	1075	510	0	13958664
7	494	676	1	13958757
8	304	606	0	13958795
9	699	411	0	13959103
...

Table 3.3 Event Data includes an *id* for positional identification, and index *i* that combines *x*, *y*, and *p*.

The first three numbers recorded in an event are the coordinates (x, y) and the polarity, p . These numbers share some properties, mainly that they are three random numbers with a limited range. For x , the range is $[0, 1279]$, y is $[0, 719]$ and p is $[0, 1]$. Because these numbers indicate the location and polarity of where an event took place, they will be known as the Index.

On the other hand, the Timestamp is always incremental. For this particular camera, the timestamp is written as an integer; however, it may be possible to find examples with floating points. A key difference regarding the Index part of the event is that these numbers cannot be predicted to be within a certain range. The maximum value will depend on how long the sensor was activated. Thus, compression strategies between the Index and the Timestamp will have differences.

3.1.2 Event Queues

Because events can number upward to the millions, a suggested data structure to keep track of them all would be a queue. Queues are dynamically sized data structures that can grow or shrink in a memory-efficient way, a distinct advantage over other structures such as arrays. Queues utilize a first-in-first-out (FIFO), where only the first element can be read and dequeued (some coding languages allow access to the event in the back, but no way to dequeue it). Queues were selected as an optimal way to store events due to their memory efficiency and low-resource utilization, as well as the iterative nature of the data compression algorithm.

3.2 Techniques and Strategies

This section details each strategy as well as the mathematical operations that the algorithm will execute. Subsections 3.2.1, 3.2.2, 3.2.3, 3.2.5 will introduce strategies needed for index header compression. In Subsection 3.2.4, timestamps will be replaced by the delta between values. Next, Subsection 3.2.6 will identify the largest event timestamps and compress them separately. For the remaining timestamps, Section 3.2.7 will reduce them by adding encoding instructions next to the data. Afterwards, Sections 3.2.8, 3.2.9, and 3.2.10 will remove bit padding from the data.

3.2.1 Index Header Data Structure

Another data class that is prevalent in the algorithm is the Index Headers. These entries will help reduce the number of index values in the events. For example, Polarity Index Header Compression (Pol) will group all the 0-valued events first, then group all the 1-valued events afterwards. Once grouped, the polarity value of each event becomes redundant, as it can be represented as the total number of events that share the same polarity.

Using both tables (3.4, 3.5), the algorithm can determine the polarity of the event by position rather than a bit value present in each event entry. For example, if each polarity can be represented using 1 bit, without index compression, it would be necessary to use 6711138bits for the Berlin Event File. On the other hand, by utilizing this Index Header Compression it is possible to represent polarity with the number of events in the Table (3.5), which only needs 44bits (22bits per written entry, *Start* can be ignored), to represent the number of elements with $p = 0$ and $p = 1$ (Initial Timestamps will only be used for delta compression).

3.2.2 Index Header Compression

The table above (3.5) shows an example index header table. The first value is the Index Type; this can have the enum values of *Start*, *X_Position*, *Y_Position*, and *Polarity*. They indicate which part of the Event Index is represented. *Start* keeps track of the overall number of events and the initial timestamp, so it will not be written in the final file.

There are eight different ways to generate the Index Header Compression Tables (3.6), each of which is different in size. However, it is important to note that not all data entries will be recorded in the file.

<i>id</i>	Index <i>i</i>			<i>t</i>
	<i>x</i>	<i>y</i>	<i>p</i>	
1	189	513	0	13958339
3	887	102	0	13958570
4	243	693	0	13958593
5	674	220	0	13958653
...
2	640	467	1	13958368
7	494	676	1	13958757
13	1143	692	1	13959217
16	334	672	1	13959481
...

Table 3.4 Events are sorted by the polarity, first $p = 0$, then $p = 1$.

Index Type	Value	Number of elements	Initial Timestamp
Start	0	6711138	13958339
Polarity	0	3355579	13958339
Polarity	1	3355559	13958368
Total entries: 3			
Written entries: 2			

Table 3.5 Polarity Index Header that indicates the number of events per Index Type

Only the entries with the Index Type of the smallest denomination will be utilized. The denomination from largest to smallest is $X_Position$, $Y_Position$, $Polarity$; therefore, the $X - Y$ Index will only keep the $Y_Position$ entries, but the $X - Pol$ Index will only keep the $Polarity$ entries.

Compression	Binary	#Entries	IndexType kept
<i>None</i>	000	0	<i>None</i>
<i>Pol</i>	001	2	<i>Polarity</i>
<i>Y</i>	010	720	<i>Y_Position</i>
<i>Y - Pol</i>	011	1440	<i>Polarity</i>
<i>X</i>	100	1280	<i>X_Positoin</i>
<i>X - Pol</i>	101	2560	<i>Polarity</i>
<i>X - Y</i>	110	921600	<i>Y_Position</i>
<i>X - Y - Pol</i>	111	1843200	<i>Polarity</i>

Table 3.6 Shows how Different Index Header Compression Algorithms will yield different lengths

Table (3.6) shows that the number of entries can vary depending on the compression selected. As such, queues will also be utilized to store the Index Headers. Here is an example queue of some illustrative data showing the X-Pol Index Header Compression.

3.2.3 Sorting Algorithm

Events must be sorted by their index values to be able to generate the Index Header Tables 3.8, and subsequently execute the Index Header Compression. Sorting the values group values with the closest predecessors and successors. A proper sorting algorithm must be selected, as execution speed is of importance.

Index Type	Value	Number of elements	Initial Timestamp
<i>Start</i>	0	6711138	13958339
<i>X_Positoin</i>	0	5242	13958339
<i>Polarity</i>	0	2618	13958339
<i>Polarity</i>	1	2624	16885230
<i>X_Positoin</i>	1	5280	14809560
<i>Polarity</i>	0	2657	15936274
<i>Polarity</i>	1	2623	14809560
<i>X_Positoin</i>	2	0	0
<i>Polarity</i>	0	0	0
<i>Polarity</i>	1	0	0
<i>X_Positoin</i>	3	5217	13996523
...

Table 3.7 Illustrative example of Index Entries for X-Pol

The Index values are both integer and positive; as such, a bucket sort would be optimal. This sorting algorithm is non-comparative. Each event can be queued into a vector array by using the index value as the memory index. An example in pseudocode is provided below in the equation.

$$\begin{aligned}
 Event &= Event_Queue.front() \\
 vector[Event.X_Value].push(Event)
 \end{aligned}
 \tag{3.1}$$

After sorting the data within a vector, events with a specified index value can be counted, and these values will be added to the Index Header entries. Of course, multiple events can have the same index value; in such cases, events will be ordered chronologically based on their timestamp.

Once the events are fully sorted, they can be returned to their original data structure with the new order. However, if the Index Header entries are still kept, it is possible to re-partition the event queue and re-sort the algorithm.

$$\begin{aligned}
 Event &= Partitioned_Event_Queue.front() \\
 vector[Event.Y_Value].push(Event)
 \end{aligned}
 \tag{3.2}$$

3.2.4 Delta Encoding

The process for delta encoding events is simple: subtract the current timestamp from the previous value. For the initial timestamp, save it apart from the rest and set the initial event timestamp to zero. For delta encoding to work optimally, the data is ordered sequentially, as all deltas will be positive. The data written in Table 3.3 shows how Timestamps are already sorted.

Because timestamps are now smaller, the number of bits necessary to represent them also shrinks. Figure 3.1 shows the size distribution. To illustrate the size of the timestamps for comparison purposes, the number of event timestamps is multiplied by the number of bits required for timestamp representation and summed.

$$\begin{aligned}
 TimestampSummationSize &= (TotalNumberOfEvents)(n_{max}) \\
 NoIndexCompressionTimestampSummationSize &= 67,111,380bits
 \end{aligned}
 \tag{3.3}$$

Where n_{max} is the maximum number of bits required for the Timestamp Representation, $n_{max} = 10$

There are complications when trying to combine Index Header Compression, the Sorting Algorithm, and Delta Encoding. Table 3.4 shows the events sorted in a different way, where all the events are sorted by polarity first and timestamp second. The first necessity that arises is to save two initial timestamps, one for the $p = 0$ events and one for the $p = 1$ events. The second necessity is to record the number of events

X-Y-Pol Index		X-Y Index		X-Pol Index	
Index Type	Value	Index Type	Value	Index Type	Value
<i>Start</i>	0	<i>Start</i>	0	<i>Start</i>	0
<i>X_Position</i>	0	<i>X_Position</i>	0	<i>X_Position</i>	0
<i>Y_Position</i>	0	<i>Y_Position</i>	0	<i>Polarity</i>	0
<i>Polarity</i>	0	<i>Y_Position</i>	1	<i>Polarity</i>	1
<i>Polarity</i>	1	<i>Y_Position</i>	2	<i>X_Position</i>	1
<i>Y_Position</i>	1	<i>Y_Position</i>	3	<i>Polarity</i>	0
<i>Polarity</i>	0	<i>Y_Position</i>	4	<i>Polarity</i>	1
<i>Polarity</i>	1	<i>Y_Position</i>	5	<i>X_Position</i>	2
<i>Y_Position</i>	2	<i>Y_Position</i>	6	<i>Polarity</i>	0
...
Total: 2764801		Total: 922881		Total: 3841	
Written: 1843200		Written: 921600		Written: 2560	

Y-Pol Index		X Index		Y Index	
Index Type	Value	Index Type	Value	Index Type	Value
<i>Start</i>	0	<i>Start</i>	0	<i>Start</i>	0
<i>Y_Position</i>	0	<i>X_Position</i>	0	<i>Y_Position</i>	0
<i>Polarity</i>	0	<i>X_Position</i>	1	<i>Y_Position</i>	1
<i>Polarity</i>	1	<i>X_Position</i>	2	<i>Y_Position</i>	2
<i>Y_Position</i>	1	<i>X_Position</i>	3	<i>Y_Position</i>	3
<i>Polarity</i>	0	<i>X_Position</i>	4	<i>Y_Position</i>	4
<i>Polarity</i>	1	<i>X_Position</i>	5	<i>Y_Position</i>	5
<i>Y_Position</i>	2	<i>X_Position</i>	6	<i>Y_Position</i>	6
<i>Polarity</i>	0	<i>X_Position</i>	7	<i>Y_Position</i>	7
...
Total: 2161		Total: 1281		Total: 721	
Written: 1440		Written: 1280		Written: 720	

<i>Polarity</i>	
Index Type	Value
<i>Start</i>	0
<i>Polarity</i>	0
<i>Polarity</i>	1
Total: 3	
Written : 2	

No Index	
Index Type	Value
<i>Start</i>	0
Total: 1	
Written: 0	

Table 3.8 Shows the different amount of Index Header Entries depending on the Index Compression Algorithm, as well as note how many will be written in the compressed file

x,y,p	t		x,y,p	t
189,513,0	13958339		189,513,0	0
640,467,1	13958368		640,467,1	29
887,102,0	13958570		887,102,0	202
243,693,0	13958593		243,693,0	23
674,220,0	13958653	→	674,220,0	60
1075,510,0	13958664		1075,510,0	11
494,676,1	13958757		494,676,1	93
304,606,0	13958795		304,606,0	38
699,411,0	13959103		699,411,0	308
...

Table 3.9 Only save the differences between timestamps, and set the initial timestamp to zero (Berlin Data)

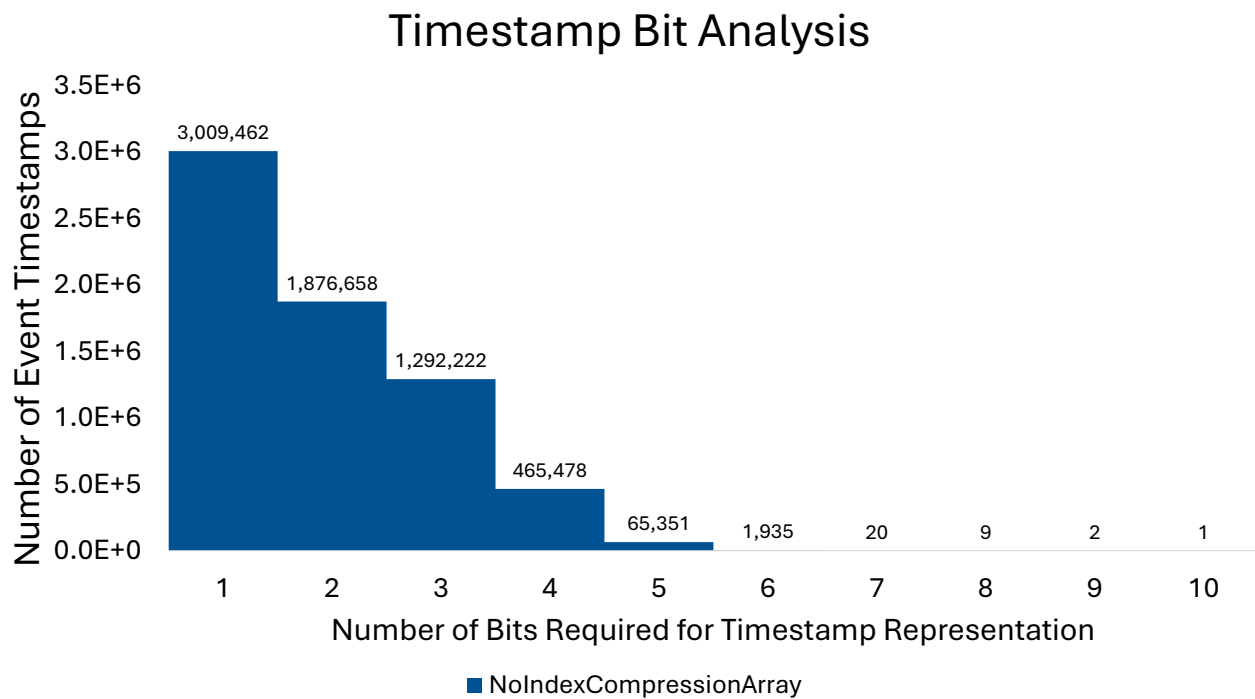


Figure 3.1 Event timestamp bit distribution for delta encoded events without any Index Header Compression, maximum timestamp is 550, and needs 10 bits to be represented.

for each entry, illustrated in Table 3.5. Afterwards, the delta compression can continue, but it will not be as efficient, as the delta between timestamps will increase, as can be observed in Figure 3.2.

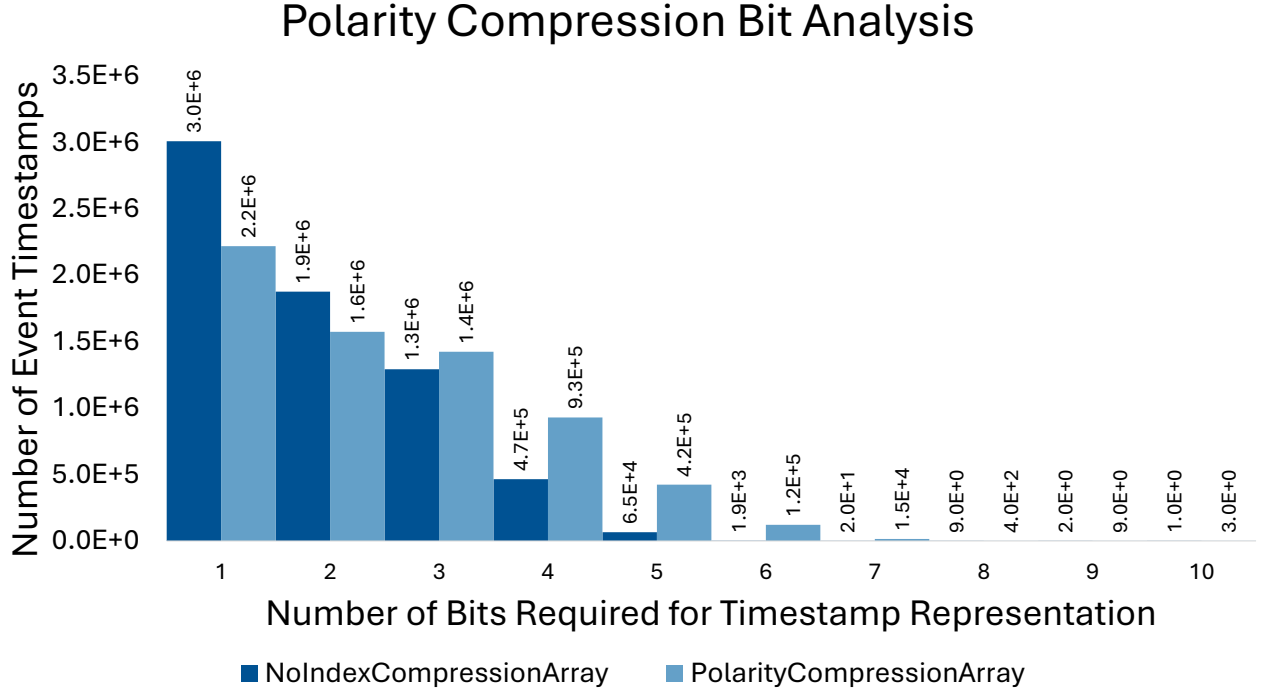


Figure 3.2 Comparison between No Index Header Compression and Polarity Index Header Compression.

To illustrate the increased size of the timestamps, equation 3.3 is applied to the Polarity Compressed values resulting in:

$$PolarityCompressionTimestampSummationSize = 73,822,518bits \quad (3.4)$$

The tradeoff between Index Header Compression and Delta encoding can be further seen if the Index is fully compressed (Table 3.3).

Figure 3.3 shows how data can be dense or sparse. For the No Index Compression case, the data is dense and is often located on the lowest number of bits. Polarity Compression is slightly more sparse, as data entries are distributed more towards higher values up to 11 bits. X Y Polarity Compression is by far the sparsest, having events from *1bit* all the way to *25bits*. This sparsity density trade-off is the result of the competing strategies of Index Header Compression and Delta Encoding.

Please note that summation sizes will change once other timestamp compression algorithms, such as dictionary and variable-length encoding, are applied.

3.2.5 Pair Encoding

To reduce the number of data entries, the first step is to Pair Encode (Subsection 2.8) the x , y , and p into a single Index number. The previous Subsection goes into detail about the Cantor Pairing Function, which is an efficient solution if the range of two numbers extends to infinity. However, the Index i values have a limited range; thus, another less computationally intensive range multiplication algorithm will be implemented. Each number is multiplied by the maximum range of the previous ones.

The step-by-step process is as follows: Obtain the values each variable can have:

$$\begin{aligned} x &\in \{0, 1, 2, \dots, 1278, 1279\}, \\ y &\in \{0, 1, 2, \dots, 718, 719\}, \\ pol &\in \{0, 1\} \end{aligned} \quad (3.5)$$

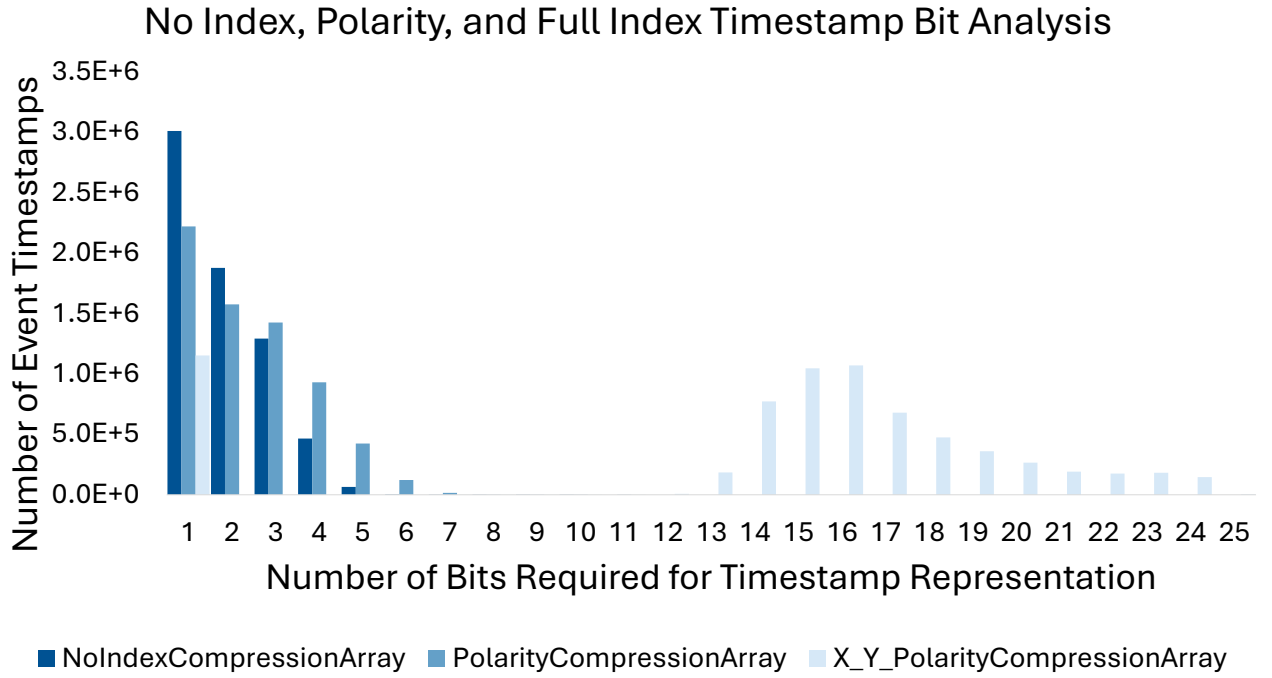


Figure 3.3 Comparison between timestamp lengths for No Index Header Compression and Polarity Index Header Compression.

Please note that some cameras may have the initial value as 1. To reduce memory constraints, subtract the minimum value from the recorded data. In addition, polarity may be represented as $pol \in \{-1, 1\}$, which is inefficient because negative values negatively impact performance. To reduce memory constraints, map polarity as instructed in 3.5. For the particular configuration of the EventSat camera, these steps will be unnecessary.

$$\begin{aligned}
 x_i &\rightarrow x_i - x_{min} \\
 y_i &\rightarrow y_i - y_{min} \\
 pol_i &= \begin{cases} -1 & \rightarrow 0 \\ 1 & \rightarrow 1 \end{cases}
 \end{aligned} \tag{3.6}$$

Where $i = \{1, 2, 3, 4, \dots\}$ is the number of the event.

When decoding, apply the inverse equation to re-obtain the original values. For the particular configuration of the EventSat camera, these steps will be unnecessary.

$$\begin{aligned}
 x_i &\rightarrow x_i + x_{min} \\
 y_i &\rightarrow y_i + y_{min} \\
 pol_i &= \begin{cases} 0 & \rightarrow -1 \\ 1 & \rightarrow 1 \end{cases}
 \end{aligned} \tag{3.7}$$

Calculate the maximum range of the variables:

$$\begin{aligned}
 x_{range} &= x_{max} - x_{min} = 1279 - 0 = 1279, \\
 y_{range} &= y_{max} - y_{min} = 719 - 0 = 719, \\
 pol_{range} &= pol_{max} - pol_{min} = 1 - 0 = 1
 \end{aligned} \tag{3.8}$$

Next, the values must be manipulated in such a way that they can be encoded and decoded into a single number without changing the original values. The following technique will order the ranges from smallest to largest, and multiply the data entry values by the previous ranges, like so:

$$\begin{aligned}
pol_i &\rightarrow pol_i \\
y_i &\rightarrow (y_i)(pol_{range}) \\
x_i &\rightarrow (x_i)(y_{range})(pol_{range})
\end{aligned}
\tag{3.9}$$

Afterwards, sum the three values to create the single number index entry:

$$index_i = x_i + y_i + pol_i \tag{3.10}$$

A preview of the results, utilizing the Berlin Data, can be seen in Figure 3.4 as well as Table 3.10.

Index		
x coordinate	y coordinate	polarity
189	513	0
640	467	1
...

Multiply index by the maximum range value + 1

$(x_i)(y_{range}+1)(pol_{range}+1)$	$(y_i)(pol_{range}+1)$	(pol_i)
$(189)(720)(2) = 272160$	$(513)(2) = 1026$	$0 = 0$
$(640)(720)(2) = 921600$	$(467)(2) = 934$	$1 = 1$
...

Sum the values together into a single Index number

Index number
$272160 + 1026 + 0 = 273186$
$921600 + 934 + 1 = 922535$
...

Figure 3.4 Visual step-by-step representation of Pair Encoding for the Berlin Data.

x,y,p	t		Index i	t
189,513,0	0	→	273186	0
640,467,1	29		922535	29
887,102,0	202		1277484	202
243,693,0	23		351306	23
674,220,0	60		971000	60
1075,510,0	11		1549020	11
494,676,1	93		712713	93
304,606,0	38		438972	38
699,411,0	308		1007382	308
...

Table 3.10 Example results of Pair Encoded Index entries for the Berlin Data.

Please note that if Index Header Compression is utilized, the efficiency of pair encoding will be reduced, as both strategies seek to compress the same data (Table 3.11).

Index Header Compression	Pair Encoding
<i>None</i>	<i>X – Y – Pol</i>
<i>Pol</i>	<i>X – Y</i>
<i>Y</i>	<i>X – Pol</i>
<i>Y – Pol</i>	<i>X</i>
<i>X</i>	<i>Y – Pol</i>
<i>X – Pol</i>	<i>Y</i>
<i>X – Y</i>	<i>Pol</i>
<i>X – Y – Pol</i>	<i>None</i>

Table 3.11 Having Index Header Compression in X, Y, and/or Polarity will result in Pair Encoding compressing the index entries that weren't utilized

3.2.6 Dictionary Encoding

The main objective of dictionary encoding is to reduce the maximum timestamp size. By observing Figure 3.1 it is clear that the events are unevenly distributed, and that most event timestamps require *4bits* or less. Implementing a method that separates long timestamps and short timestamps could reduce the overall size of the compressed event file.

This section will be divided into mathematical explanations, followed by the graphical result at the end.

The first step for calculating the thresholds for long and short timestamps is to observe what happens to the event entries to measure the threshold. A Dictionary Event id entry with an arbitrary minimum bit size of *5bits* would appear as so:

Dictionary Events			Non-Dictionary Events	
<i>id</i>	<i>i</i>	<i>t</i>	<i>i</i>	<i>t</i>
3	1277484	202	273186	0
5	971000	60	922535	29
7	712713	93	351306	23
8	438972	38	1549020	11
9	1007382	308	1538946	25
...

Table 3.12 Events that are larger than the arbitrarily chosen value of 5 bits (values higher than 31) will be categorized as Dictionary Events, and stored in the file with their Id value

The Dictionary Events will need to be stored with the *id* value to indicate their location with the Non-Dictionary Events. The Index *i* remains the same for both events. Timestamp *t* for Dictionary Events would still need the maximum size of *10bits* (3.1), but Non-Dictionary Events will only need *5bits*, reducing the size of the timestamps by half. The number of *5bits* was arbitrarily chosen to illustrate an example

To accurately select a threshold value, a histogram analysis must be conducted. This analysis will calculate the sizes of the timestamps depending on the threshold. The trade-off in the analysis is the summation of the Dictionary Event Timestamps vs. the Non-Dictionary Event Timestamps. The results are graphed at the end of this subsection.

The formulas for the analysis are as follows:

First, the events from Figure 3.1 are arranged within an event vector by bit size:

$$\mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \dots \\ e_{n-1} \\ e_n \end{bmatrix} = \begin{bmatrix} 3009462 \\ 1876658 \\ \dots \\ 2 \\ 1 \end{bmatrix} \quad (3.11)$$

Where e is the Event distribution array, n is the maximum timestamp size for this particular case, 10
Afterwards, e will be utilized to generate the event summation array.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_n \end{bmatrix} = \sum_{k=i}^n e_k = \begin{bmatrix} \sum_{k=1}^n e_k \\ \sum_{k=2}^n e_k \\ \sum_{k=3}^n e_k \\ \dots \\ \sum_{k=n}^n e_k \end{bmatrix} = \begin{bmatrix} 6711138 \\ 6711137 \\ 6711135 \\ \dots \\ 3009462 \end{bmatrix} \quad (3.12)$$

Where x is the Event Summation Distribution array, i indicates the current index of the vector x , k indicates the values of e that will be in the summation, and n indicates the size of the maximum timestamp

Next, a vector to calculate the size of the uncompressed Non-Dictionary Event Timestamps by multiplying vector x by the size of the remaining timestamp size:

$$\mathbf{b} = \begin{bmatrix} n \\ n-1 \\ n-2 \\ \dots \\ 1 \end{bmatrix}, \mathbf{w} = \mathbf{x} \times \mathbf{b} = \begin{bmatrix} 6711138 \times 10 \\ 6711137 \times 9 \\ 6711135 \times 8 \\ \dots \\ 3009462 \times 1 \end{bmatrix} = \begin{bmatrix} 67111380 \\ 60400233 \\ 53689080 \\ \dots \\ 3009462 \end{bmatrix} \quad (3.13)$$

Where b is a vector that goes from the maximum n down to 1, and w is the size of the uncompressed Non-Dictionary Event Timestamps in bits.

The next step is to calculate the bits of the compressed Dictionary Events. The easiest way to calculate is by subtracting the total number of events from vector x and multiplying by n :

$$\mathbf{v} = (\mathbf{x}_1 - \mathbf{x}) \times (n + s) \quad (3.14)$$

$$\mathbf{v} = \begin{bmatrix} x_1 - x_1 \\ x_1 - x_2 \\ x_1 - x_3 \\ \dots \\ x_1 - x_n \end{bmatrix} \times (n + s) = \begin{bmatrix} 6711138 - 6711138 \\ 6711138 - 6711137 \\ 6711138 - 6711135 \\ \dots \\ 6711138 - 3009462 \end{bmatrix} \times 33 = \begin{bmatrix} 0 \\ 10 \\ 30 \\ \dots \\ 37016760 \end{bmatrix} \quad (3.15)$$

Where v is the size of the Dictionary Events and s is the size in bits of the number of events (for Berlin data, the number of events is 6711138 and $s = 23$).

With the size of the remaining uncompressed events and the size of the compressed events, the final step is to find out the minimum number of bits:

$$\mathbf{d} = \mathbf{w} + \mathbf{v} = \begin{bmatrix} 67111380 + 0 \\ 60400233 + 33 \\ 53689080 + 99 \\ \dots \\ 3009462 + 122155308 \end{bmatrix} = \begin{bmatrix} 67111380 \\ 60400266 \\ 53689179 \\ \dots \\ 125164770 \end{bmatrix} \quad (3.16)$$

Afterwards, search the vector for the minimum value m :

$$d = \begin{bmatrix} 67111380 \\ 60400266 \\ 53689179 \\ 46978278 \\ 40267692 \\ 33610766 \\ \textcolor{red}{28796774} \\ 36117294 \\ 69997834 \\ 125164770 \end{bmatrix} \rightarrow \text{bitsize} = \begin{bmatrix} 10 \\ 9 \\ 8 \\ 7 \\ 6 \\ 5 \\ \textcolor{red}{4} \\ 3 \\ 2 \\ 1 \end{bmatrix}, m = 4 \quad (3.17)$$

$$\text{event_timestamp_bitsize} \begin{cases} > m & \rightarrow \text{DictionaryEvent} \rightarrow (n + s) \\ \leq m & \rightarrow \text{NonDictionaryEvent} \rightarrow m \end{cases} \quad (3.18)$$

Therefore, the maximum bit size for non-dictionary events will be 4bits ; if a number needs 5bits or larger, the event will be dictionary encoded. Figure 3.5 shows the analysis graphically.

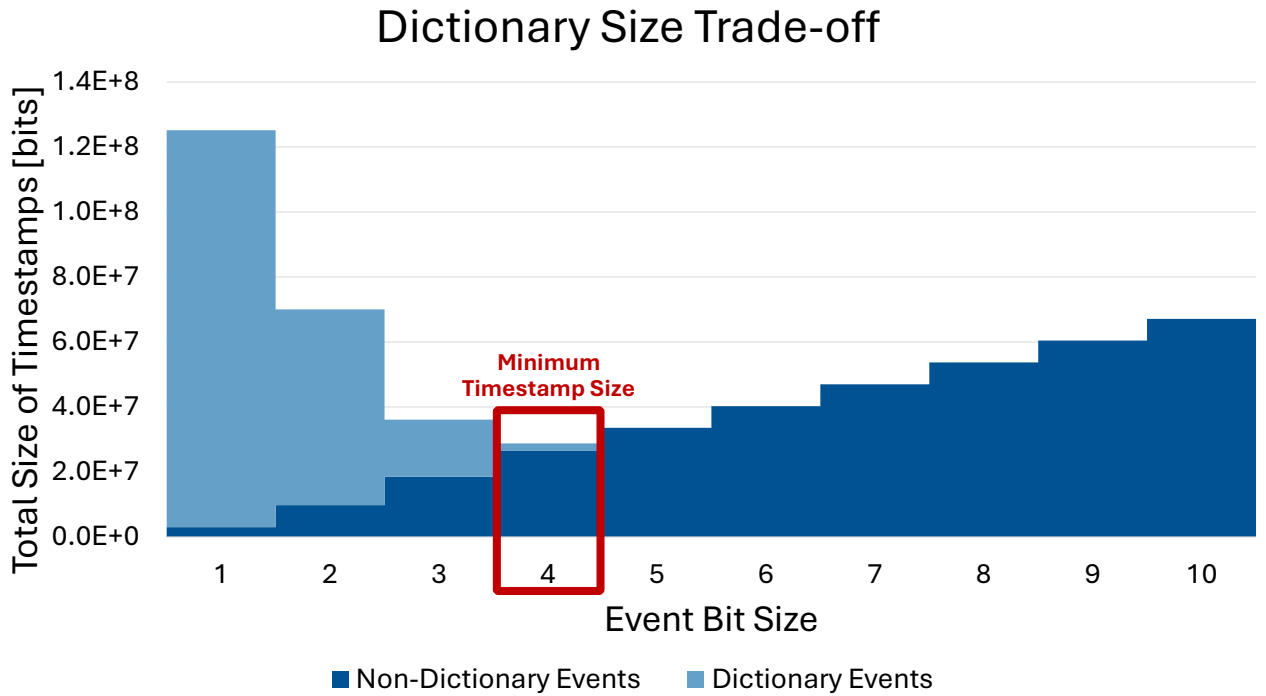


Figure 3.5 The optimal value for the dictionary cut off is 4bits ; therefore, all events that are 5bits or over should be dictionary compressed

3.2.7 Variable Length Encoding

Variable Length Encoding is a simple algorithm in which a trigger bit $[0, 1]$ is added to each timestamp, indicating whether the timestamp should have a long or short memory size. A histogram analysis is performed to find the optimal value for the Variable Length Encoding. It is also possible that the result will be a larger size. In such cases, it is best to avoid the variable-length encoding altogether. For the Berlin dataset, the size is 28796774, which was calculated in eq 3.17.

This section will be divided into mathematical explanations, followed by the graphical result at the end.

The example data was obtained from the Berlin file.

First, a vector is created with the leftover events that were not compressed with dictionary encoding.

$$\mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \dots \\ e_{m-1} \\ e_m \end{bmatrix} = \begin{bmatrix} 3009462 \\ 1876658 \\ 1292222 \\ 465478 \end{bmatrix} \quad (3.19)$$

Here, m is the maximum number of bits for the uncompressed events. This result was obtained at the end of the dictionary compression.

A summation of the vector is calculated:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_n \end{bmatrix} = \sum_{k=1}^m e_k = \begin{bmatrix} \sum_{k=1}^m e_k \\ \sum_{k=2}^m e_k \\ \sum_{k=3}^m e_k \\ \dots \\ \sum_{k=m}^m e_k \end{bmatrix} = \begin{bmatrix} 6643820 \\ 6178342 \\ 4886120 \\ 3009462 \end{bmatrix} \quad (3.20)$$

Afterwards, the number of events is multiplied by the length of the timestamp plus the additional bit of the trigger.

$$\mathbf{b} = \begin{bmatrix} m \\ m-1 \\ m-2 \\ \dots \\ 1 \end{bmatrix} + 1, \mathbf{w} = \mathbf{x} \times \mathbf{b} = \begin{bmatrix} 6643820 \times 5 \\ 6178342 \times 4 \\ 4886120 \times 3 \\ 3009462 \times 2 \end{bmatrix} = \begin{bmatrix} 33219100 \\ 24713368 \\ 14658360 \\ 6018924 \end{bmatrix} \quad (3.21)$$

Where w is the total size of the Short Timestamp Events in bits.

Next, the size of the long Timestamps events:

$$\mathbf{v} = (\mathbf{x}_1 - \mathbf{x}) \times (m + 1) \quad (3.22)$$

$$\mathbf{v} = \begin{bmatrix} x_1 - x_1 \\ x_1 - x_2 \\ x_1 - x_3 \\ \dots \\ x_1 - x_n \end{bmatrix} \times (m + 1) = \begin{bmatrix} 6643820 - 6643820 \\ 6643820 - 6178342 \\ 6643820 - 4886120 \\ 6643820 - 3009462 \end{bmatrix} \times 5 = \begin{bmatrix} 0 \\ 2327390 \\ 8788500 \\ 18171790 \end{bmatrix} \quad (3.23)$$

Afterwards, both the Short Timestamp Events w and the Long Timestamp Events v are summed to find the total event size vector

$$\mathbf{d} = \mathbf{w} + \mathbf{v} = \begin{bmatrix} 33219100 + 0 \\ 24713368 + 2327390 \\ 14658360 + 8788500 \\ 6018924 + 18171790 \end{bmatrix} = \begin{bmatrix} 33213100 \\ 27040758 \\ 23446860 \\ 24190714 \end{bmatrix} \quad (3.24)$$

$$\mathbf{d} = \begin{bmatrix} 33213100 \\ 27040758 \\ \mathbf{23446860} \\ 24190714 \end{bmatrix} \rightarrow \text{bitsize} = \begin{bmatrix} 4 \\ 3 \\ \mathbf{2} \\ 1 \end{bmatrix}, h = 2 \quad (3.25)$$

Finally, the minimum value must be compared to the original size, to verify that the variable length indeed reduces file size by using the result in Eq. 3.17 ($23446860 < 28796774$).

$$\text{EventTimestampBitsize} \begin{cases} > h \rightarrow \text{LongTimestamp} + \text{Trigger} \rightarrow m + 1 \\ \leq h \rightarrow \text{ShortTimestamp} + \text{Trigger} \rightarrow h + 1 \end{cases} \quad (3.26)$$

As such, events whose timestamps can be represented with h or fewer bits will be assigned the Short Event Timestamp h , while the rest will have the original timestamp size m .

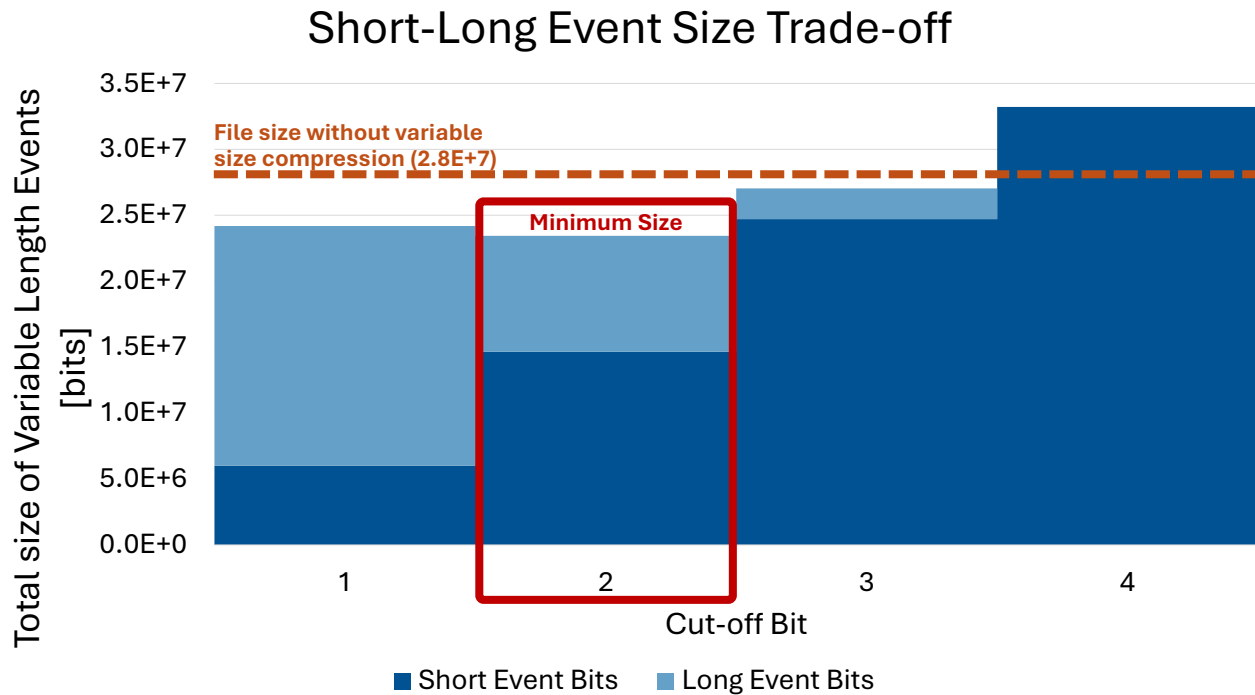


Figure 3.6 The optimal value for the short timestamp cut off is *2bits*; therefore, all events that are *3bits* or over should be long timestamp

3.2.8 Unsigned Integers

Typically, integers in computer languages have positive and negative values. To represent negative values, two's complement is utilized. Essentially, one bit indicates if the number is negative; if so, the polarity of the rest of the bits is flipped. Since all of the numbers in the Event File remain positive (if polarity is negative, it can be set to 0), unsigned integers could be used to reduce the size of each int utilized by 1.

3.2.9 Custom Integer Length

Normally, integers assign a limited amount of memory for a single number. The typical int sizes in bits are *int8*, *int16*, *int32*, *int64* (unsigned). If the number of bits necessary to represent a number is known, an integer length that is more data efficient can be selected.

Data Name	Largest Entry	# Bits Needed	Best U_Int
Pair Encoded Index	1,843,200	25	<i>int32</i>
Dictionary Timestamp	550	10	<i>int16</i>
Long Variable Timestamp	15	4 + 1(trigger)	<i>int8</i>
Short Variable Timestamp	3	2 + 1(trigger)	<i>int8</i>
Number of Events Entry	6,711,138	27	<i>int32</i>

Table 3.13 Shows how smaller numbers could be assigned to smaller integers.

Please note that most programming languages have a small number of integer sizes. Additionally, the minimum size is *int8*, as the basic unit of memory is a byte. Afterwards, all additional sizes are 8^n where $n = [1, 2, 3, 4...]$. As such, unless another strategy, such as Bit Splicing, is implemented, a significant amount of memory space will be redundant.

3.2.10 Bit Splicing

Most computing systems can only store values in Bytes (*8bits*), but as Table 3.13 shows, some entries can be different than a number or multiple of 8. In such cases, a strategy exists where several variables can be split and stored in bytes without padding between values.

Bit Splicing Customized Length Int Into Bytes Without Padding

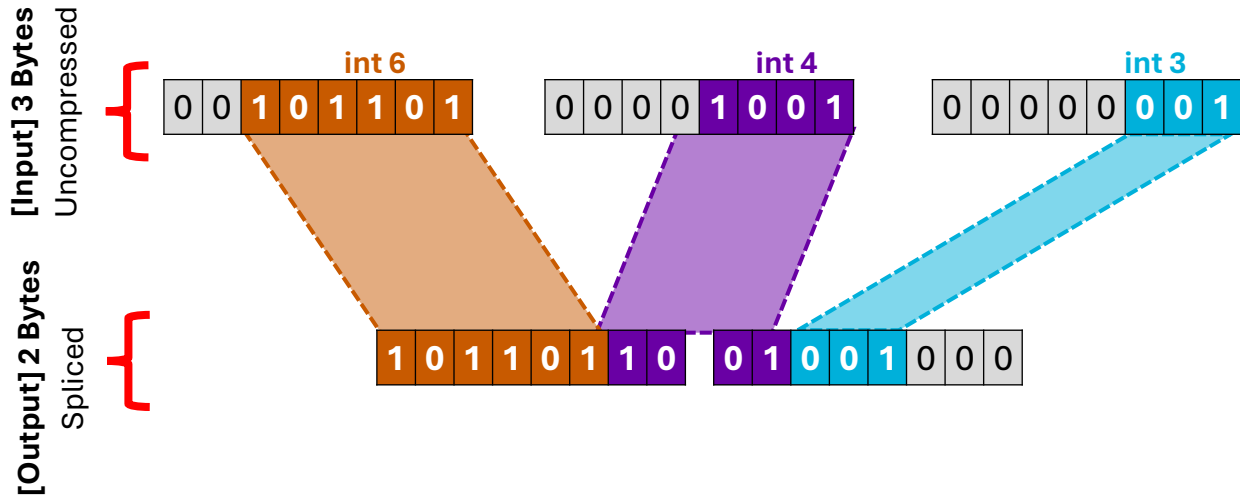


Figure 3.7 Data is always stored in bytes, but using bit splicing, padding can be removed, and data can be efficiently stored from *3bytes* to *2bytes*.

Data must be manipulated at the bit level; the C++ Bitwise Operators are utilized (Table 3.14). The first four operators function the same as logic gates *AND*, *OR*, *XOR*, and *COMPLEMENT*. These are executed per bit in a byte (or larger) variable. The last two operators are used to shift the bits. When bits are shifted, if the edge values are replaced or discarded, careful consideration is needed to avoid losing information.

Bit Operator Symbol	Operation	Explanation
&	<i>AND</i>	If both 1, result 1, else 0
	<i>OR</i>	If any 1, result 1, else 0
^	<i>XOR</i>	If both different, result 1, else 0
~	<i>COMPLEMENT</i>	Flips 0 to 1, and vice versa
<<	<i>ShiftLeft</i>	Shift bits to the left
>>	<i>ShiftRight</i>	Shift bits to the right

Table 3.14 Shows the bit operators that are used for manipulating bytes.

3.3 Techniques and Strategies: Summary

The relevant techniques and strategies have all been introduced in both the Literature Review (2) and the Methodology (3). Table 3.15 is a summary of which strategies will be employed to compress which part of

the event data detailed in Table 3.3. Lastly, Table 3.16 shows how each encoding strategy cooperates with other strategies.

Strategy Name	Index $i(x, y, p)$	Timestamp t
Index Header Compression	Optional	No
Delta Encoding	No	Yes
Dictionary Encoding	No	Yes
Variable Length Encoding	No	Yes
Unsigned Integers	Yes	Yes
Custom Integer Length	Yes	Yes
Bit Splicing	Yes	Yes
Pair Encoding	Yes	No
Sorting Algorithm	For compression	For decompression

Table 3.15 Shows if a compression strategy will be used for Index or Timestamp compression.

	Sorting Algorithm	Pair Encoding	Bit Splicing	Custom Integer Length	Unsigned Integers	Variable Length Encoding	Dictionary Encoding	Delta Encoding
Index Header Compression	O	X	O	O	O	--	--	X
Delta Encoding	X	--	O	O	O	O	O	
Dictionary Encoding	--	--	O	O	O	X		
Variable Length Encoding	--	--	O	O	O			
Unsigned Integers	O	O	O	O				
Custom Integer Length	--	O	O					
Bit Splicing	--	O						
Pair Encoding	--							

Table 3.16 Shows how some strategies have a Positive [O] interaction with others, Neutral with [--], or Negative[X]

Most of the negative interactions in Table 3.16 are due to the Index Header Compression, the Sorting Algorithm, and the Delta Encoding. Header compression requires data to be sorted; however, if data is sorted, delta encoding will not be as efficient, as the differences between measurements will increase. Additionally, Delta Encoding has a positive interaction with Variable Length Encoding and Dictionary encoding, as data is concentrated into a denser histogram. As such, the Index Header Compression and Sorting Algorithms will remain deactivated. The last negative interaction is between Dictionary Encoding and Variable Length Encoding. Choosing one of these will result in the other being less efficient. However, if Variable Length is used after Dictionary Encoding, it is still possible to compress the data further; therefore, both algorithms will be enabled.

Additionally, Unsigned Integers, Custom Integer Length, and Bit Splicing have either Neutral or Positive interactions. These strategies help reduce padding by manipulating bytes, rather than modifying data. As such, space is used more efficiently.

3.4 Algorithm Structure

The algorithm will be split into two different sections, Compression 3.4.1 and Decompression 3.4.2. When the algorithm is executed, the user can choose one of the two modes. As such, they are separated in different subsections.

3.4.1 Compression

Compression begins by reading a .raw (Ev3/Ev2) file or a common text file such as .txt or .csv. The strings are then processed into numerical values for x , y , polarity, and timestamp.

The next step is to generate the Id Events, which combine the index values of x , y , and p into a single number and additionally include a positional id . The algorithm then checks if there will be an index i header compression, which reduces the index size of the Id Event (by default set to no index compression). If there is an index compression, the events will be bucket-sorted, a type of non-comparative sorting algorithm, utilizing the x , y , and p as "key values" and merely distributing the events to a vector array (of queues). As such, values do not need to be compared with each other. Afterward, delta encoding can begin, where only the difference between the timestamps is recorded. For index header compression, the sorted events will be parallelized, and delta encoding will only be used for events within the specified index. Finally, x , y , and polarity will be pair encoded into a single index value (unless there was index header compression, in which case some values will not be encoded).

Once the data entries are transformed to Id Events, the histogram analysis will be run to provide the most efficient thresholds for compression (Table 3.8). First, the index analysis calculates the number of index entries that will be written in the compressed file. Then, a dictionary histogram analysis will calculate the optimal threshold to reduce the maximum timestamp for a certain percentage of Id Event entries (equation 3.26). Next, the events with reduced timestamps will be further compressed using variable-length encoding. The events are then divided into short timestamps and long timestamps.

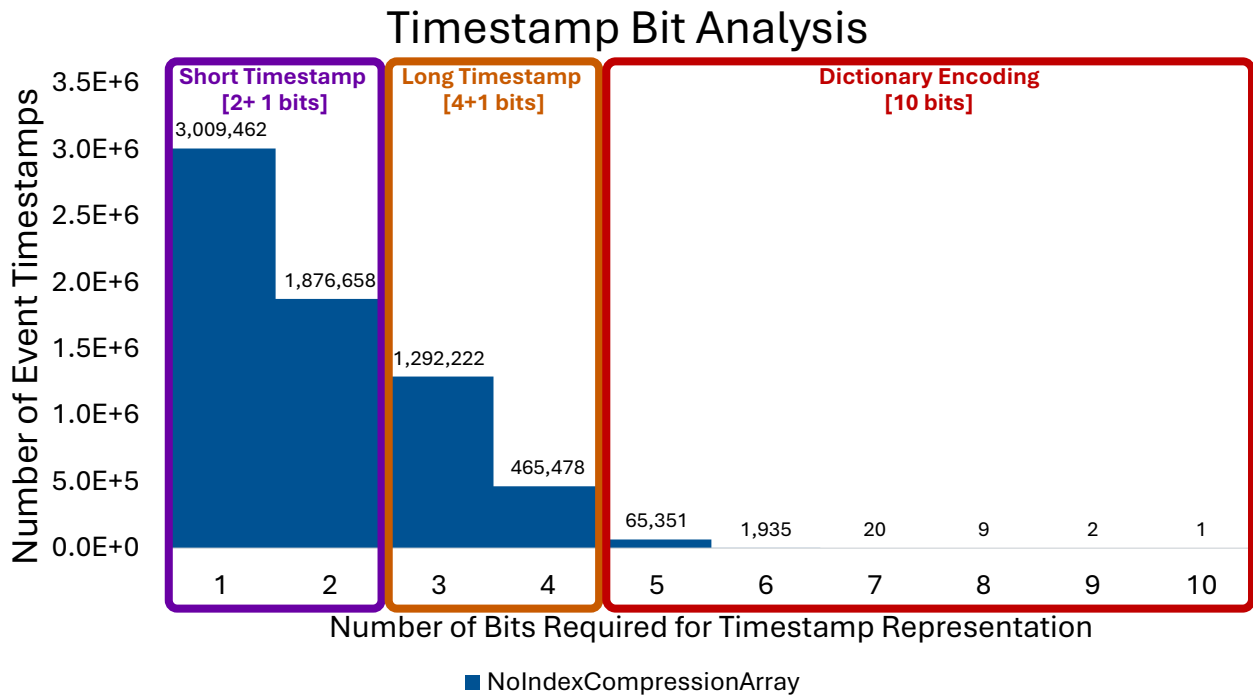


Figure 3.8 Shows the thresholds calculated by the histogram analysis for the compression algorithm using the Berlin dataset.

The final step is to prepare the data for the file creation. The events will be split into long longs; afterwards, these will be bit-spliced into chars. Each bit will be fully loaded in the byte; once a byte has 8 bits,

a new byte will be added. Once the char list is finished, a few values will be added to the compressed file header. When writing the file, the compressed header will be transformed into a string indicating index, dictionary events, and finally the variable-length events. The user will input the filename at the moment that the compression is executed.

3.4.2 Decompression

To decompress, the file itself must provide specific instructions for decompression. As such, the first step will be to re-obtain the encoding instructions from the file header. The header will include information about the camera geometry, initial timestamp, number of events, number of chars for each specific compression algorithm, etc. With these instructions, the algorithm can begin reading and correctly interpret the rest of the characters in the file. The character queue will be split into an index, variable-length, and dictionary character queues.

The chars will be transformed back into the original data structures. Chars will be Bit-spliced to build back the long longs; afterwards, the long longs will be interpreted back as integers. Index entries need the number of events and initial timestamps. Dictionary-encoded events will have ids, index, and timestamp, while the variable-length events will have a trigger value for memory size, index, and timestamp. Dictionary events will be inserted where their Ids indicate. Once events are in chronological order, the timestamps will be delta-decoded using the initial timestamps.

A problem arises if there is Index Compression, as events could have the same timestamps. Because events were initially sorted by timestamps, if two events have the same timestamp, the code cannot differentiate which one was written first without adding more overhead information. Once the index-sorted events are re-sorted into a chronological order, a systematic bias is introduced, as events with a smaller index will always appear first. The files are virtually the same, as each event's integrity remains intact; however, the overall integrity of the file is not maintained, as a systematic bias is introduced.

The events are then written into the final file format .csv or .txt; these file types are the easiest to read and manipulate. Once the file is decompressed, the final responsibility is fulfilled. It is possible to add future file-checking procedures, such as hash integrity checks, to verify that the file was indeed the same.

4 Results

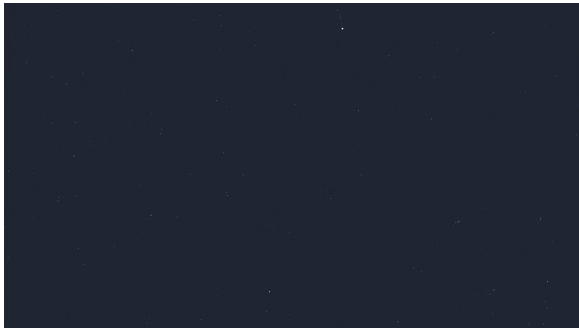
The Results section will delve into all the outputs of the compression algorithm as well as statistics. First, Section 4.1 will present the test files used for compression. Next, Section 4.2 will mention additional index compression strategies that were not implemented in the final version due to file integrity errors. Then, Section 4.3 will show how each algorithm component contributes to the overall compression.

4.1 Compression Sizes and Datasets

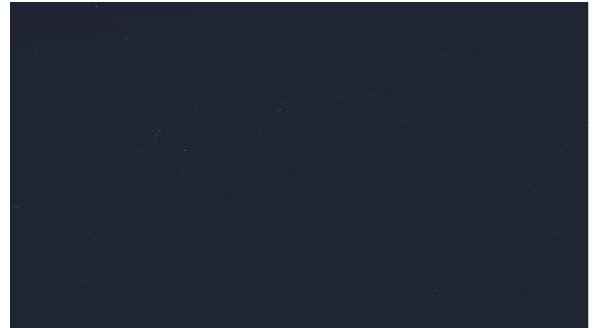
For a viable result, the EventSat lossless compression algorithm presented in this thesis must create files smaller than the initial input, while maintaining data integrity by running a file comparison command (FC in Windows, diff in Linux). As a benchmark, other compression algorithms will be run to measure the performance. The results will be evaluated using the compression ratio (a higher ratio means better results):

$$CompressionRatio = 1 - \frac{CompressedFileSize}{UncompressedFileSize} \quad (4.1)$$

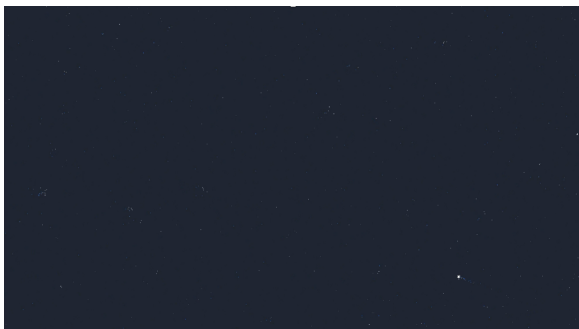
To analyze the performance of the algorithm, four scenarios will be utilized: Berlin, Spin, Madrid, and London.



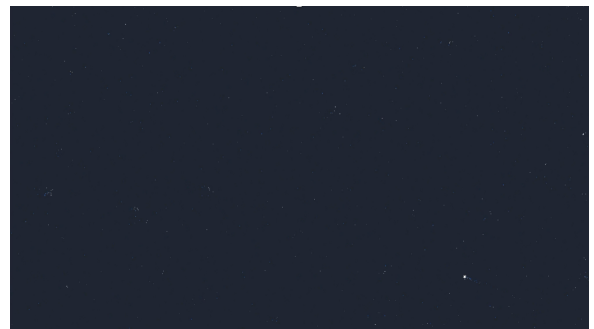
(a) Berlin



(b) London



(c) Madrid



(d) Spin

Figure 4.1 Event Frame Captures of each of the four files considered

The first two are full recordings with the expected length of an observation. The other two are additional datasets that are of shorter length. Berlin is to be considered the main overall result of the algorithm. Even

File Name	File Size (MB)	Scenario Length (s)	Event Rate (MEvs)
Berlin	130.1	43.44	9.1
London	5.0	13.57	9.1
Madrid	8.1	24.77	9.1
Spin	170.0	27.01	9.1

Table 4.1 Key Characteristics of each of the four scenarios considered. *MEvs* is million events per second

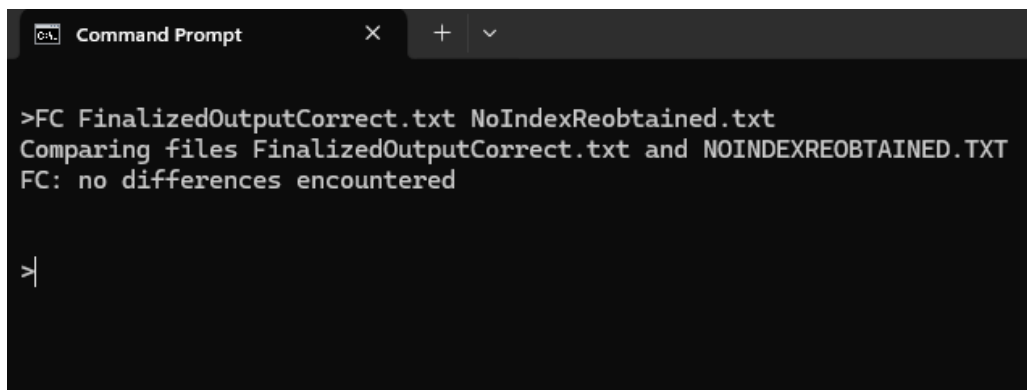
though the files do show what the camera would observe in space, the main limitation was having only four test files. Additional entries could further show performance in different scenarios.

The resulting compressed file from the algorithm [Bin] is evaluated together with the two other compression algorithms: [Raw], the default compressor for the camera, and [Zip], a common compression file for Windows. Applying Eq. 4.1 to the data yields Table 4.2.

File Sizes [bytes]	Berlin Dataset	Spin Dataset	Madrid	London
Human Readable	133,177,219	173,610,480	8,250,704	5,121,357
Raw	31,577,649	42,355,357	7,841,209	4,418,273
Zip	42,575,962	57,257,994	2,958,836	1,834,221
Bin (ours)	21,019,730	28,167,826	1,530,245	934,837
Compression Ratios [%]	Berlin Dataset	Spin Dataset	Madrid	London
Raw	76.29	75.60	4.96	13.73
Zip	68.03	67.02	64.14	64.18
Bin (ours)	84.22	83.78	81.45	81.75

Table 4.2 Shows if a strategy will be used for Index or Timestamp compression.

To verify that the algorithm compressed correctly, a file integrity test is run. The FC Windows command is then executed, Figure 4.2. Since no file differences were detected, compression is considered a success as no data is lost.



```

C:\> FC FinalizedOutputCorrect.txt NoIndexReobtained.txt
Comparing files FinalizedOutputCorrect.txt and NOINDEXREOBTAINED.TXT
FC: no differences encountered
>|

```

Figure 4.2 Correct integrity output, original and compressed-decompressed file are the same.

The EventSat compression algorithm file [Bin] was smaller than the [Raw, Zip] files across all datasets. As such, the compression algorithm can be considered as a viable alternative to the commonplace compression algorithms. There are additional data procedures that will affect file size and downlink capabilities, such as packet headers and packet transmission management. Thus, the full downlink-observation ratio cannot be calculated within the scope of this thesis alone.

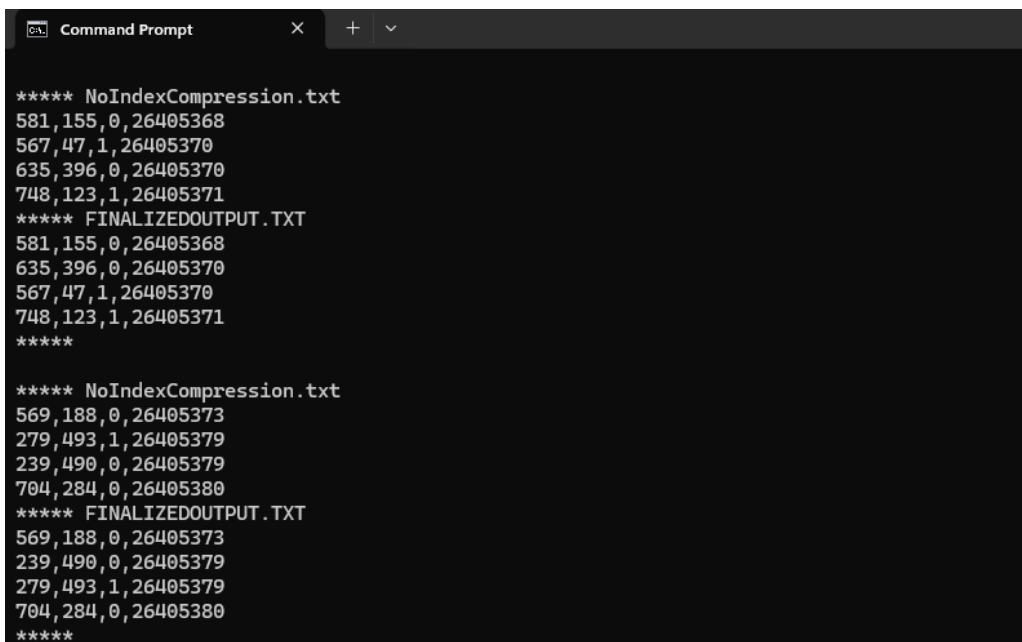
4.2 Changing Index Compression Settings

An investigation was conducted for an alternative formulation of our algorithm that involved index compression. Events are sorted into their corresponding buckets, associating value with location. The index part of an event requires less data to be represented, but needs an additional header entry indicating how many events share the same index 3.8. Delta compression also becomes inefficient as data has been re-sorted 3.3. Table 4.3 compares the impact on file size for different index compression approaches. Polarity Compression results in a slightly smaller compressed file. However, an issue arises when two events have the same timestamp; in such cases, the algorithm introduces a systemic bias. The index would influence whether the next event appears first in the decompressed file once the events are re-sorted from index to timestamp. As a result, file verification tests and decompression would be significantly more difficult. To simplify on-board operations, this thesis will continue the analysis with no index compression.

Table 4.3 also includes Y Polarity and X Y Polarity. These compression algorithms increase file size in two ways. The first is the additional index entries Table 3.8, which increases file overhead. The second is larger deltas in between time measurements (Figure 3.3), making delta encoding overall worse. As such, they are not considered viable for the final algorithm configuration.

Index Compression Algorithm	File Size [bytes]	File Size Comparison %
No Index Compression	21,019,730	100
Polarity Compression	20,871,010	99.3
Y Polarity Compression	21,226,766	101
X Y Polarity Compression	28,301,401	135

Table 4.3 Shows that Index compression can result in a slightly smaller file.



```
***** NoIndexCompression.txt
581,155,0,26405368
567,47,1,26405370
635,396,0,26405370
748,123,1,26405371
***** FINALIZEDOUTPUT.TXT
581,155,0,26405368
635,396,0,26405370
567,47,1,26405370
748,123,1,26405371
*****

***** NoIndexCompression.txt
569,188,0,26405373
279,493,1,26405379
239,490,0,26405379
704,284,0,26405380
***** FINALIZEDOUTPUT.TXT
569,188,0,26405373
239,490,0,26405379
279,493,1,26405379
704,284,0,26405380
*****
```

Figure 4.3 Shows incorrect integrity validation, systematic bias results in the same timestamp events switching order.

4.3 Ablation Studies

This section examines the effectiveness of each compression strategy as it is implemented on the data. The analysis visualizes the effectiveness of the compression strategies. These studies will not consider Index Compression Algorithms as they have already been discussed in Section 4.2.

Table 4.4 shows the data format at each step of the event compression. In the first step, the data is transformed from characters to integers. Secondly, Dictionary encoding separates timestamps with more bytes and an additional *id* value. Variable Encoding further splits the Non-Dictionary events. However, due to the small difference in timestamp size, the events remain the same length in bytes. The final three strategies allow for events to be smaller in bits, rather than bytes, resulting in a significant size reduction. Event distribution refers to the percentage of events that are either dictionary, long, or short events. Filesize is the expected size of an observation with 6,000,000 recorded values.

Data Format	Individual Event Sizes	Event Distribution ¹ [%]	File Size ¹ [MB]
Raw String	18, 19, 20 bytes	100	109
Event as Integers	16 bytes	100	91
Dictionary Encoding	Dictionary: 11 bytes	1	39.7
	Non-Dictionary: 7 bytes	99	
Variable Encoding	Dictionary: 11 bytes	1	39.7
	Long Events: 7 bytes	73	
	Short Events: 7 bytes	26	
UnsignedIntegers/ Custom Integer Length/ Bit Splicing	Dictionary: 54 bits	1	19.13
	Long Events: 27 bits	73	
	Short Events: 25 bits	26	

Table 4.4 Shows the Ablative Study of compression strategies. 1. Expected File Size from a dataset with 6e6 events

The combination of all the strategies in files that are approximately 20% of the original file size. This result is due to the hybrid approach between the dictionary and variable-length encoding. The histogram analysis (Section 3.2.6) showed effective results in obtaining the minimum thresholds for encoding data. This approach automatically calculates the compression thresholds, removing the need for fine-tuning general-purpose compression algorithms. Furthermore, bit-splicing removed an additional 20MB of data padding, so only relevant data is encoded into the file. This final strategy enables the hybrid approach to work more efficiently.

5 Conclusions

This subsection will go over the conclusions of the thesis. First, a summary of the thesis and the research contributions. Section 5.1 talks about limitations regarding test files and testing integration with other flight software for the mission. Next, Section 5.2 mentions extensions that can be added to the work presented in this thesis. Lastly, Section 5.3 presents final arguments for the thesis.

This thesis presents a lossless compression algorithm for event data capable of reducing file size by 80%. The algorithm was developed specifically for EventSat, a CubeSat mission for the TUM Chair of Spacecraft Systems. The algorithm manages to incorporate dictionary encoding for the longest 1% of events and variable-length encoding to separate the remaining short 26% and long 76% event sizes. Histogram analysis was conducted to calculate the optimal threshold for encoding strategies, $5bits$ for the dictionary and $3bits$ for variable-length long events. Additionally, bit splicing permitted data to be more compact by $20MB$ because data can be written on a bit-level. All the while considering low-resource architecture data structures, such as queues, that can be executed on board and on the ground.

5.1 Limitations

Due to the collaborative structure of satellite missions and the strict timeframe of the thesis, certain limitations were inevitable. The first limitation is the lack of dataset diversity. Only four files were compressed using the algorithm. More varied test datasets could present more accurate results. The second limitation is the lack of real-time usage and integration with other flight software. At the time of writing, mission software is still in development; full integration testing is performed afterwards.

5.2 Future Work

Several possible additional extensions to this work could further improve the performance of the proposed algorithm. These strategies may result in smaller file sizes. The efficiency of other encoding strategies may be impacted, as lossless data can only be compressed up to a certain point.

The first strategy is Huffman encoding, where values are assigned variable-length binary codes [17]. This could be implemented on the Index portion of the events; index values that are repeated often would be associated with shorter codes, resulting in smaller file sizes. The main trade-off is the Huffman table, which matches the compressed values with index values, as well as the repetition of the index value within the file.

The second strategy is to implement decoding instructions within timestamps. This would be useful in dictionary events by replacing the *id* value. Timestamps would be the only element that remains in the dictionary portion of the file, while the index of the event would be in place. The index is unchanged, but the timestamp is replaced with a coded value (such as the maximum binary value 11111_{bin}). The decoder would then look at the dictionary timestamps and retrieve the value during decompression.

5.3 Final Conclusions

A successful lossless compression algorithm must understand the data on a deeper level to compress a file and maintain its integrity. Once data generation is understood, a correct approach to compressing information can be selected. For example, event information is only positive integers and commas to

separate values. Further analysis reveals that x , y , and p are randomly distributed, while timestamps t are sequential.

The context of operation is relevant for algorithm design. For satellites, downlink capabilities are significant. Therefore, strategies such as bit-splicing can be beneficial.

With these considerations, a lossless compression algorithm for space operations was developed. The results from test datasets showed that the algorithm was efficient for compressing data for downlink. The algorithm outperformed the generic default camera compression algorithm and the default Windows zip file. All the while, file integrity is maintained.

Bibliography

- [1] Goel Abhay, Sharma Abhishek, and Gupta Namita. A variant of bucket sort shell sort vs insertion sort. In *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–5, 2019.
- [2] Keshav Bajpai and Ashish Kots. Implementing and analyzing an efficient version of counting sort (e-counting sort). *International Journal of Computer Applications*, 98(9):1–2, 2014.
- [3] Martin Burtscher and Paruj Ratanaworabhan. gfpcc: A self-tuning compression algorithm. In *2010 Data Compression Conference*, pages 396–405, 2010.
- [4] S Chowdhury, SR Bhadra Chaudhuri, and CT Bhunia. Dynamic dictionary based online compression technique applicable to mobile computing. In *National Conference, NUCONE*, volume 6, pages 449–453.
- [5] Soumit Chowdhury, Amit Chowdhury, S.R. Bhadra Chaudhuri, and C.T. Bhunia. Data transmission using online dynamic dictionary based compression technique of fixed & variable length coding. In *2008 International Conference on Computer Science and Information Technology*, pages 930–936, 2008.
- [6] Wikimedia Commons. Comparison computational complexity, 2017.
- [7] Sydney Dolan. Payload configuration document updated aug11. Technical University of Munich Caroline Herschel Straße 100/II 85521 Ottobrunn - Germany, 2025. Technical University of Munich Chair of Spacecraft Systems. Internal documentation.
- [8] Sydney Dolan, Lara Schuberth, Rugved Arge, R. M. G. Alarcia, Vincenzo Messina, C. J. J. Oliver, Francesco Salmaso, Jaspar Sindermann, Federico Sofio, and Alessandro Golkar. Design and analysis of an event camera payload for space-based object detection on the eventsat 6u cubesat mission. In *Proceedings of the 15th IAA Symposium on Small Satellites for Earth System Observation*, 2025.
- [9] Sasidhar Duggineni. Impact of controls on data integrity and information systems. *Science and technology*, 13(2):29–35, 2023.
- [10] Adolf Fenyi, Michael Fosu, and Bright Appiah. Comparative analysis of comparison and non comparison based sorting algorithms. *International Journal of Computer Applications*, 2020.
- [11] Tim Flohrer and Holger Krag. Space surveillance and tracking in esa's ssa programme. In *7th European conference on space Debris*, volume 7, 2017.
- [12] Kinshuk Goel, Palak Dwivedi, and Oshin Sharma. Performance analysis of various sorting algorithms: Comparison and optimization. In *2023 11th International Conference on Intelligent Systems and Embedded Design (ISED)*, pages 1–5, 2023.
- [13] Oussama Harkati, Lemnouar Noui, and Assia Beloucif. Image encryption via qsd decomposition and pairing functions: A novel approach. In *2024 1st International Conference on Innovative and Intelligent Information Technologies (IC3IT)*, pages 1–5, 2024.
- [14] John A Kennewell and Ba-Ngu Vo. An overview of space situational awareness. In *Proceedings of the 16th International Conference on Information Fusion*, pages 1029–1036, 2013.

- [15] Joella Lobo and Sonia Kuwelkar. Performance analysis of merge sort algorithms. In *2020 International Conference on Electronics and Sustainable Communication Systems (ICESC)*, pages 110–115, 2020.
- [16] Marcellino Marcellino, Davin William Pratama, Steven Santoso Suntiarko, and Kristien Margi. Comparative of advanced sorting algorithms (quick sort, heap sort, merge sort, intro sort, radix sort) based on time and memory usage. In *2021 1st International Conference on Computer Science and Artificial Intelligence (ICCSAI)*, volume 1, pages 154–160, 2021.
- [17] Janarbek Matai, Joo-Young Kim, and Ryan Kastner. Energy efficient canonical huffman encoding. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 202–209, 2014.
- [18] Vladimir N Potapov. Redundancy estimates for the lempel–ziv algorithm of data compression. *Discrete Applied Mathematics*, 135(1-3):245–254, 2004.
- [19] B Reddaiah. A study on pairing functions for cryptography. *International Journal of Computer Applications*, 975:8887, 2016.
- [20] Hridoy Roy, Md. Shafiuzzaman, and Md. Samsuddoha. Srcs: A new proposed counting sort algorithm based on square root method. In *2019 22nd International Conference on Computer and Information Technology (ICCIT)*, pages 1–6, 2019.
- [21] Prophesee S.A. Raw file format. https://docs.prophesee.ai/stable/data/file_formats/raw.html. Accessed: 2025-10-13.
- [22] Waseem Shariff, Mehdi Sefidgar Dilmaghani, Paul Kieilty, Mohamed Moustafa, Joe Lemley, and Peter Corcoran. Event cameras in automotive sensing: A review. *IEEE Access*, 12:51275–51306, 2024.
- [23] Mahima Singh and Deepak Garg. Choosing best hashing strategies and hash functions. In *2009 IEEE International Advance Computing Conference*, pages 50–55, 2009.
- [24] Haoliang Tan, Zhiyuan Zhang, Xiangyu Zou, Qing Liao, and Wen Xia. Exploring the potential of fast delta encoding: Marching to a higher compression ratio. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 198–208. IEEE, 2020.
- [25] Kun Xiao, Pengju Li, Guohui Wang, Zhi Li, Yi Chen, Yongfeng Xie, and Yuqiang Fang. A preliminary research on space situational awareness based on event cameras. In *2022 13th International Conference on Mechanical and Aerospace Engineering (ICMAE)*, pages 390–395, 2022.
- [26] You Yang, Ping Yu, and Yan Gan. Experimental study on the five sort algorithms. In *2011 Second International Conference on Mechanic Automation and Control Engineering*, pages 1314–1317, 2011.
- [27] Bhavani Yerram and Jaya Krishna Bhonagiri. An efficient sorting algorithm for binary data. In *2020 11th International Conference on Computing, Communication and Networking Technologies (ICC-CNT)*, pages 1–4, 2020.